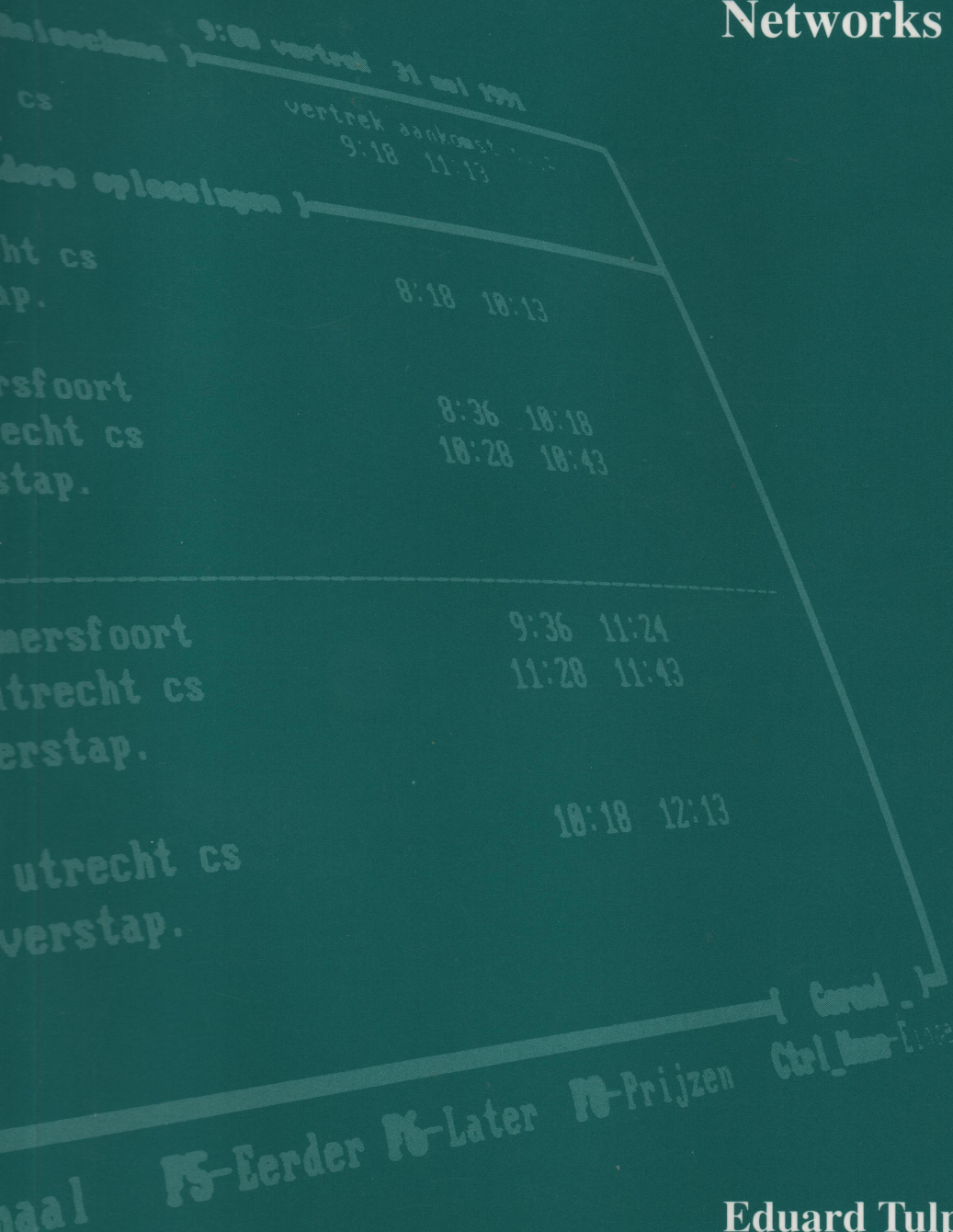
Searching Time-table Networks



Eduard Tulp

Stellingen

Behorende bij het proefschrift "Searching Time-table Networks" van Eduard Tulp

(1) Een discreet dynamisch netwerk is door zijn fundamentele eigenschappen een doelmatige representatie van een dienstregelingsnetwerk.

(2) Het DYNET algorithme is een correcte en efficiënte manier om een discreet dynamisch netwerk te doorzoeken.

(3) SRM is, met name in combinatie met DYNET*, een goede techniek om de efficiëntie van het doorzoeken van een dienstregelingsnetwerk te verhogen.

(4) Een reizigersinformatiesysteem voor dienstregelingsgestuurd vervoer moet actief gedrag vertonen.

(5) Door een gebrek aan goede informatie wordt het openbaar vervoer slecht benut. Het gebruik van goede reizigersinformatiesystemen zal leiden tot een bewustere keuze van vervoermiddel, en een verhoogd gebruik van het openbaar vervoer.

(6) Het gebruik van een reizigersinformatiesysteem als stuurmiddel mag niet leiden tot een vermindering van de kwaliteit van de informatie.

(7) De basis van dynamische reizigersinformatie is niet de actuele situatie, maar een voorspelling van de situatie voor de komende uren.

(8) Indien bij een automatiseringsproject "de gebruiker centraal wordt gesteld", dan mag dit niet betekenen dat de specificatie van het project aan de gebruiker wordt overgelaten.

(9) Hoe gebruikersvriendelijker een programma is, des te slechter wordt de handleiding gelezen.

(10) Om de Nederlandse taal te moderniseren, verdient het aanbeveling om de uitdrukking "Loopt als een trein" ter vervangen door "Loopt als een Hoge Snelheids Trein (HST)".

(11) De werken van Bach worden, ondanks waarschuwingen van Mozart, in een te hoog tempo uitgevoerd. Waarschijnlijk is dit te wijten aan het feit dat veel musici het *alla breve* negeren, en de mogelijkheden van klassieke instrumenten overschatten.

(12) In de hedendaagse muziek worden musici te veel beoordeeld op de beheersing van de techniek, en te weinig op het vermogen om de emotie van de muziek over te brengen.

(13) Net zoals bij kunstmatige inseminatie, is bij kunstmatige intelligentie de natuurlijke variant veel opwindender.

VRIJE UNIVERSITEIT

Searching Time-table Networks

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. C. Datema,
hoogleraar aan de faculteit der letteren,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der economische wetenschappen en econometrie
op donderdag 31 oktober 1991 te 15.30 uur
in het hoofdgebouw van de universiteit, De Boelelaan 1105

door

Eduard Tulp

geboren te Voorburg

Drukkerij Elinkwijk BV, Utrecht
1991

Promotoren : prof. dr. A.A.I. Holtgrefe
prof. dr. L. Siklóssy
Referenten : prof. dr. ir. R. Hamerslag
dr. M.H. Slagmolen

Searching Time-table Networks

This book was printed by Drukkerij Elinkwijk BV, Utrecht, September 1991. Reproduced from camera-ready copy, which was typeset using Ventura Publisher Professional. Layout and all illustrations by the author. The maps were digitized by Ch. Keuken. Cover by NS-Design.

Copyright © 1991 by Eduard Tulp

All rights reserved. No part of this book may be reproduced in any form or by any means without permission in writing from the author.

Contents

Contents v

Preface xi

1. Introduction 1

2. Discrete Networks 7
2.1. A quick introduction to graph theory 7
2.1.1. A weighted non-directed graph 7
2.1.2. A weighted directed graph 8
2.2. Representing a railway network by a graph 8
2.3. Representing a railway service network by a graph 9
2.4. Representing waiting time 10
2.5. A discrete network 11

3. Searching A Graph 15
3.1. Shortest path algorithms 15
3.1.1. Matrix algorithms: Floyd's algorithm 15
3.1.2. Tree building algorithms 16
3.1.2.1. Label correcting algorithms: Moore's algorithm 17
3.1.2.2. Label setting algorithms: Dijkstra's algorithm 18
3.1.2.3. An improvement of Dijkstra's algorithm 19
3.2. Remembering the route of the shortest path 20
3.3. The dynamic programming principle 21
3.4. Searching bidirectionally 21
3.5. Conclusion 22

4.	**Searching Discrete Networks**	**23**
4.1.	Adapting Dijkstra's algorithm to discrete networks	23
4.2.	The principle of optimality for discrete networks	24
4.3.	An example	25
4.4.	Relevant edges	25
4.5.	Suboptimality of solution	27
4.6.	An optimal path in a discrete network	27
4.7.	Traversing the suboptimal solution	27
4.8.	Finding the optimal solution: the second pass	28
4.9.	Using the results from the first search	29
4.9.1.	A two-pass algorithm: DISNET	32
5.	**Discrete Dynamic Networks**	**35**
5.1.	Visiting costs	35
5.2.	Platform dependency	35
5.3.	Train dependency	36
5.4.	A discrete dynamic network	37
5.5.	Space requirements of the CON function	39
6.	**Searching Discrete Dynamic Networks**	**41**
6.1.	Adapting the search algorithm to discrete dynamic networks	41
6.2.	A counter example: the effect of CON	43
6.3.	The invalidity of the Markovian property	44
6.4.	The principle of optimality for discrete dynamic networks	45
6.5.	Relevant edges in discrete dynamic networks	46
6.6.	A search algorithm for discrete dynamic networks: DYNET	47
6.6.1.	The correctness of the forward pass	49
6.6.2.	The correctness of the backward pass	50
6.7.	Dynamic generation of vertices	51
7.	**General Dynamic Networks**	**53**
7.1.	An example application	53
7.2.	Dynamic networks	54
7.3.	Searching a dynamic network	54
7.3.1.	The principle of optimality for dynamic networks	55
7.3.2.	The correctness of the algorithm	57
7.3.3.	Dynamic generation of vertices	59

8. Space Reduction Method 61
8.1. Domains of application 61
8.2. The algorithm for SRM 64
8.3. The different steps in SRM 64
8.3.1. The Idealized Skeleton Graph ISG 65
8.3.2. The Idealized Solution 65
8.3.3. Loosening the solution in ISG 66
8.3.4. The Reduced Graph G' 67
8.3.5. The search in the Reduced Graph G': first pass 68
8.3.6. Verification of Optimality 68
8.3.7. Repair: second pass 70
8.4. The choice of the coefficient p 70
8.5. Application 71

9. Heuristic Search 73
9.1. The A* search algorithm 73
9.2. Admissibility of A* 74
9.3. Consistent heuristics 74
9.4. A* as a modified Dijkstra algorithm 76
9.5. Using the results from SRM in A* 76
9.6. Using heuristics in DYNET: DYNET* 77

10. Offset Vertices 81
10.1. Offset vertices 81
10.2. Offset vertices in undirected graphs 82
10.3. Offset vertices in directed graphs 83
10.3.1. Transforming a graph 83
10.3.2. The algorithm to transform a directed, weighted graph 86
10.3.3. Adapting Dijkstra's algorithm to offset vertices 87
10.3.3.1. An offset vertex as the starting vertex 87
10.3.3.2. An offset vertex as the terminating vertex 88
10.3.3.3. An offset vertex as starting vertex and terminating vertex 88
10.3.4. The Dijkstra algorithm for searching a graph with offset vertices 89
10.4. Offset vertices in discrete dynamic networks 91
10.4.1. Transforming a discrete dynamic network 92
10.4.2. The algorithm to transform a discrete dynamic network 95
10.4.3. Searching a discrete dynamic network with offset vertices 97

10.4.3.1. An offset vertex as the starting vertex; the forward pass 98
10.4.3.2. An offset vertex as the terminating vertex; the forward pass 98
10.4.3.3. Offset vertices as starting and terminating vertex; the forward pass 99
10.4.3.4. An offset vertex as the starting vertex; the backward pass 100
10.4.3.5. An offset vertex as the terminating vertex; the backward pass 100
10.4.3.6. Offset vertices as starting and terminating vertex; the backward pass 101
10.4.4. DYNET for searching a discrete dynamic network with offset vertices 102

11. Train Changes **107**
11.1. A train in a discrete dynamic network 107
11.2. A train change in a discrete dynamic network 108
11.3. Solutions with unnecessary train changes 108
11.4. Eliminating unnecessary train changes 109
11.4.1. Eliminating a train change 109
11.4.2. Preserving optimality of solution 110
11.4.3. Preserving legality of solution 110
11.4.4. The algorithm to eliminate unnecessary train changes 111
11.5. Suboptimal solutions with fewer train changes 112
11.5.1. Competing solutions 112
11.5.2. Using a change value 113
11.5.3. Using macro operators 115
11.5.3.1. Which paths to remember 116
11.5.3.2. Which paths to develop 117
11.5.3.3. The DYNET algorithm, using macro operators 117
11.5.3.4. Other cases 119

12. Implementation **123**
12.1. Representing a network 123
12.1.1. Matrix representation 123
12.1.2. Ladder representation 124
12.1.3. Forward star representation 125
12.1.4. Sorted forward star 126
12.2. Implementing the frontier 127
12.2.1. Sorted list 128
12.2.2. Binary heap 128
12.2.3. Address calculation 130
12.2.4. Circular address calculation 131
12.2.5. Address calculation with buckets 132

12.3. Implementation of TRAINS 133

13. TRAINS, An Active System 135
13.1. The theory of active systems: discontinuities 135
13.2. The necessity of active behaviour 136
13.3. The dimensions of a topic 136
13.4. Active behaviour 136
13.4.1. Subject focusing 137
13.4.2. The initial answer 137
13.4.3. Searching for alternate solutions 137
13.5. Discontinuity conflicts 137
13.6. User models 138
13.7. The application of TRAINS 138
13.8. The user model in TRAINS 138
13.9. Relevant solutions 139
13.10. An example 140

14. TRAINS, Results 143
14.1. The program 143
14.1.1. The techniques used 143
14.2. The network 144
14.3. The example questions 144
14.3.1. Heemskerk to Amsterdam CS 144
14.3.2. Hoorn to Den Haag HS 145
14.3.3. Den Haag CS to Blerick 146
14.3.4. Vlissingen to Zwolle 147
14.4. Performance 149
14.4.1. Computational effects 149
14.4.1.1. DYNET* versus SRM 151
14.4.1.2. Limiting the backward search 151
14.4.2. Time requirements 151

15. TRAINS, A Product 175
15.1. Travel information and TRAINS 175
15.2. TRAINS as a tool at enquiry offices 176
15.2.1. Introducing TRAINS at enquiry offices 177
15.2.2. The effects of professional use of TRAINS 179

15.2.2.1. Speed 179
15.2.2.2. Quality 180
15.2.2.3. Consistency 180
15.2.2.4. Regulations 180
15.2.2.5. Awareness 181
15.3. TRAINS as a commercial product 181
15.3.1. Releasing TRAINS 181
15.3.1.1. The product 182
15.3.1.2. Price 183
15.3.1.3. Packaging 183
15.3.1.4. Promotion and distribution 183
15.3.1.5. Personnel 184
15.3.2. Sales and effects 184
15.3.3. Alternate use 185
15.4. The future 185

16. Further Developments **187**
16.1. Representing other forms of public transportation 187
16.2. Discontinuities in public transportation services 187
16.2.1. The human solution: maps 188
16.2.2. The computer solution: digital maps 188
16.2.2.1. Estimating walking distances 189
16.2.2.2. Choosing the stops 189
16.2.2.3. Special objects 189
16.2.2.4. Obstacles 190
16.2.2.5. The advantage of using geographical information 190
16.3. Combining time-tables and geographical information 190
16.4. Using TRAINS in the time-table planning process 190

References **191**

Summary **197**

Samenvatting **201**

Preface

No thesis is ever the work of one single person. Some people contribute to the research represented by the thesis, some people contribute to the writing of the thesis, some people make sure that an environment for research exists, some people support and sometimes help by just being there. This thesis, of course, is no exception. Perhaps even more so since the research described in this thesis has largely been a private endeavour. This research has not been officially supported, nor has it been part of a university research program. However, two organizations, and numerous people have helped in ways large and small. This preface acknowledges them.

First of all, I would very much like to thank Laurent Siklóssy. I am greatly indebted to him. Without his encouragement, faith, advice and help there would be no thesis. It has been a privilege to work with him. Apart from his expert knowledge of artificial intelligence and research, I have benefited from his advice on writing, publishing, presenting, travel, business, and jurisdiction. I shall carry his advice and his stories with me for the rest of my life. It fills me with me great pleasure and pride that he is my *promotor*.

Close second, I would like to thank Guus Holtgrefe. Affiliated both with the Free University of Amsterdam, and the Dutch railways NS, he has played a crucial role in making it possible to put my work to practice. He introduced me to NS and its subsidiary company Centrum Voor Informatieverwerking (CVI), and was most helpful in transferring my system TRAINS to these companies. He made sure that I got a suitable job at CVI and allowed me to work things out my way, and try out my ideas in practice. There is no doubt in my mind that without his efforts, and sometimes even protection, there would not have been an *NS Reisplanner*. I hope that my efforts in making the NS Reisplanner a success have been rewarding. In his quality of professor at the economics faculty of the Free University of Amsterdam, he provided me with a friendly faculty when I needed one. As a *promotor* he proved to be an invaluable addition to Laurent Siklóssy and took care of many

administrative and practical matters. I would also like to take the opportunity to thank Mrs. Holtgrefe. I must have spoilt numerous evenings and weekends for her.

I would like to thank Ruud Hamerslag and Martin Slagmolen for their enthusiastic reception of my work, their useful remarks and encouragement.

Throughout the past years, my brother Wim has always been a great help and support as a brother, friend and colleague. His work converting and preparing time-table data, and programming silly things like user interfaces, has always been greatly underestimated. He has always accepted a less prominent role in the background without envy. Still, he co-authored the TRAINS *system*. I am proud that his name is also on NS Reisplanner. Together, we have become the infamous *Tulp brothers*.

At NS, Gijs Klomp has been a true supporter from the start. In his own bold and inimitable way he made it possible to introduce the TRAINS system at NS information centers and subsequently release it for public use. I am looking forward to continue working with him putting a system for the complete Dutch public transportation network to use. I would like to thank Gert van 't Wout for his faith and honesty in discussions. I would like to thank Peter Groenen for sharing his years of experience and time-table expertise, for his cooperation from the very beginning at NS, and for being an honest supporter. Thanks to Ineke Renkema who turned the TRAINS system into the beautifully presented product NS Reisplanner. I would also like to thank Marc Blasband, Jan Datema, and Harry van Straalen for their contributions preparing the release of the NS Reisplanner. The ladies and occasional gentlemen at the NS information centers in Hengelo, The Hague and Utrecht have been the best users I could have hoped for. Specifically, I would like to thank Mr. G. Kolkhuis Tanke, Mr. G. Nijmeijer, Ms. P. Romijn, Ms. K. van Son, Ms. L. Visser, Ms. J. Weggemans, and of course, Ms. M. van Es.

Many people at CVI have been very supportive and encouraging. First of all, the marketing communications group at CVI adopted my thesis and generously supported its preparation and printing. My friend and colleague Marcel van Heerwaarden has been a very good and minute proof reader, and made many useful suggestions. Michiel Verhoef has been great company on conference trips. Ben Gruben made sure that I need not worry about administrative fuss. CVI management consisting of Ben Gruben, Theo van Grunsven, Louis Roes and Martin Slagmolen made sure that I could do my work the way I think is best, and enabled me to go to important conferences to present my work and update my knowledge. Joop Braber did a good job making the TRAINS system a successful CVI promotional gift: the first NS Reisplanner. Many thanks to Jos Tevel who expertly guided me

through the last phase of getting a doctorate and preparing the thesis for printing, and took care of many practical matters.

Last, but in no way least, I would like to thank my brothers, sisters and parents for their support and interest. The support of my family has been important, and the encouragement of my parents has been the most important reason for me to go through with the doctor's degree.

1. Introduction

Searching is one of the fundamental areas of research both in Operations Research and in Artificial Intelligence. However ingenious a representation for a problem may be, usually some and sometimes a great deal of search is still required to find solutions to the problem. Early in the history of AI (see for example [Ni, 1971]), search problems constituted a major part of the interests of AI researchers. More recently, search has lost some of its favour, probably for two reasons. First, fast and sometimes optimal algorithms have been developed for many areas. When no such algorithms were found, it was either because the area had not been considered, or because no such algorithms were or could be found (for example, the optimal algorithms were slow). Second, some general search techniques, as implemented for example in expert systems shells, have been viewed as adequate.

In Operations Research, in the sixties and early seventies much attention was paid to the efficiency and memory requirements of search algorithms (see, for instance, [Po, 1960], [Gi, 1973], [Pa, 1974]). Due to the increase in computational power (in terms of speed and memory size) in the following years, the importance of these topics diminished. However, since the advent of relatively small and slow micro computers, again most work in the area of search algorithms concentrates on optimizing implementations of existing algorithms (see, for instance [Di, 1979], [De, 1979], [Gl, 1984], [Pa, 1984], [Vu, 1988]).

We became interested in searching railway service networks, i.e. typically finding optimal train connections from railway station s to station t, leaving at or after some start time T_{start}. An optimal solution makes us arrive at our destination as early as possible (departing after or at our planned earliest departure time T_{start}), and given this earliest arrival time, will allow us to leave as late as possible. Practice shows that people often have great difficulty planning a journey from one station to another using conventional railway guides. When the distance between the two stations is long and the journey consists of several stages, planning a journey requires searching multiple time-tables in parallel. Once a route is found, usually little

attempt is made to improve the found solution or to find a better one at a different time.

Previous attempts at searching for a quickest route in a public transportation network were made for planning purposes (see for instance [Cl, 1972], [Ga, 1984]). Never before had a system been designed to give specific information about travelling possibilities in an existing transportation network. When a system is used for capacity planning, or in passenger flow models, an approximation of the (mean) travel time is sufficient. However, when the goal is to give specific travel information, a much higher level of detail is necessary. When searching for exact travelling possibilities, the exact times of departure and arrival must be used and the correct connectional margin must be observed (a connectional margin is the time needed to change trains). This high level of detail requires much computer storage and power, and may have discouraged previous attempts. In addition, in the past, computers with sufficient capacity may have been too expensive for these customer service applications. The present availability of relatively cheap micro computers with sufficient capacity, and the increase in importance of customer oriented applications, have made the development of a travel information system possible.

We have found that using a conventional graph representation of a railway service network is not satisfactory. To represent such a network adequately we have developed the concepts of a discrete network and of a discrete dynamic network. In a discrete network there are only finite, discrete, predetermined possibilities for moving from one vertex to another. Rather than representing the discrete nature of the connections by a function giving the (varying) travel time and wait time of a connection (see [Co, 1966]), the connections themselves are made discrete. In a discrete dynamic network, in addition, visiting a vertex has a cost (possibly zero), which may depend both on the past and future route of the path through the vertex. Furthermore we introduce dynamic networks, which lack the discreteness of connections, but in which visiting a vertex has a cost.

We describe search algorithms for finding optimal paths in discrete, discrete dynamic and dynamic networks. We show that in a discrete and in a discrete dynamic network, due to the discrete nature of the connections, the definition of an optimal path must be adapted. In order to find such an optimal path, with Dijkstra's algorithm ([Di, 1959]) in mind, we have developed a two-pass algorithm. Due to the varying visiting costs in a discrete dynamic and in a dynamic network, the Markov independence (see, for instance [Hi, 1986] or [Wi, 1984]) of optimal solutions is no longer true. Therefore none of the traditional shortest path algorithms could be used (for an overview of these algorithms see for instance [De, 1984], [Dr, 1969], [Go,

1976], [Po, 1960], [Vl, 1978]). We have adapted the two-pass algorithm for searching a discrete network to handle discrete dynamic networks.

The algorithm for searching discrete dynamic networks has been implemented in a working system (TRAINS) which searches the entire Dutch railway service network. TRAINS is in current use at the Dutch railway company NS (Nederlandse Spoorwegen) and was recently introduced to the general public. Various AI techniques (symmetries, abstraction spaces, distance estimates, etc.) are used to improve the performance of TRAINS.

Although the optimal or quickest solution is thoroughly defined, it is far less clear what is the *best answer* to a user's question. In practice, it turns out that users usually overspecify their question and that this question is seldomly definite. There are many factors which determine the 'best' answer, and most users cannot even make all of these factors explicit. In the domain of travel by train, it is known that the number of train changes is important, but there may be additional factors contributing to the best answer. Furthermore, these factors may differ from case to case. Therefore, it is not possible to define the best answer in terms of goals and constraints. In order to find the best answer we cannot just have chosen to use such techniques as multiple-objective shortest path techniques (see for instance [Wh, 1982], [Wa, 1987]), or techniques to find suboptimal paths for each objective (see for instance [Ka, 1982], [Pe, 1986], [Sh, 1976], [Sh, 1979], [Ye, 1971]). Instead, we search for a number of optimal solutions, and suboptimal solutions with fewer train changes, and use a general "common sense" user model to select all relevant solutions for a user. The user decides which solution is best for her.

We now give a short overview of the thesis.

In **chapter 2**, after a quick introduction to graph theory, we show why a conventional graph is not well suited to represent a network of railway services. We introduce a discrete network which can represent a railway service network adequately by its discrete connections.

In order to find a basis for a search algorithm for discrete networks, in **chapter 3** we review graph search techniques.

With Dijkstra's graph search algorithm as a basis, in **chapter 4** we present a search algorithm for discrete networks.

In **chapter 5** we introduce visiting costs to discrete networks. The result is a discrete dynamic network in which the visiting cost for a specific vertex, incurred by the incoming and outgoing edge, is given by a connection function.

Using the algorithm for searching a discrete network as a basis, we present an algorithm for searching a discrete dynamic network in **chapter 6**.

Visiting costs are not restricted to discrete networks. They may also occur in ordinary (weighted) graphs. In order to represent visiting costs in a weighted graph adequately, in **chapter 7** we propose a dynamic network. We also give a search algorithm for dynamic networks.

In order to increase search efficiency we have developed the Space Reduction Method, which is presented in **chapter 8**. In SRM first solutions in a simpler search space, called the abstraction space, are considered in order to cut parts of the entire search space. We show how SRM can be applied to searching a discrete dynamic network.

In **chapter 9** we describe how heuristics can be used to further improve the efficiency of search algorithms. We describe how the results from SRM can be used in an A* type of extension to the algorithm for searching discrete dynamic networks.

An excellent way to decrease the amount of search necessary to find a solution, is to make sure that the network that is being searched is a small as possible. In **chapter 10** we describe how some vertices can be removed from the network when they are neither the source, nor the goal vertex. We show how the algorithm for searching discrete dynamic networks can be adapted to deal with these 'hidden' vertices.

In any practical situation, people often want not only the quickest route, but also routes with as few train changes as possible. In **chapter 11** we describe how the quickest route can be optimized for train changes, and how some (suboptimal) longer routes with fewer train changes can be found.

In **chapter 12** we look at some implementation issues, such as storage techniques and efficient implementation of the search algorithms.

Most techniques described in this thesis have been implemented in the TRAINS system, which is introduced in **chapter 13**. TRAINS is being used at NS information centers and was recently released to the general public as an electronic alternative to the conventional (paper) time-tables. TRAINS exhibits active behaviour. We describe how answering a user's request, the system will not only present the optimal solution, but also other solutions which may be of interest to the user.

In **chapter 14** we give some figures about TRAINS, the networks it searches, its performance, and some examples of questions and the solutions TRAINS finds.

In **chapter 15** we describe how TRAINS was introduced to its users, how it was adapted to their wishes, and how it was adapted for public use and finally released. We also describe some of the effects the TRAINS system and its release have had on the NS organization and how it may very well become the basis for future NS time-table information systems.

Recently, the TRAINS system was extended and other forms of public transportation were added. In order to ensure high quality information we had to adapt the representation of transportation services and extend the active component. In **chapter 16** we describe these further developments.

2. Discrete Networks

2.1. A quick introduction to graph theory

First we shall give a quick introduction to graph theory. We shall restrict ourselves to those topics which are necessary for a good understanding of the theoretical issues to be discussed. For a more lengthy and thorough introduction to graph theory and graph algorithms, we would like to refer to [Ev, 1979].

2.1.1. A weighted non-directed graph

A *graph* $G = (V, E)$ is a structure which consists of a set of vertices $V = \{v_0, v_1, \ldots\}$ and a set of edges $E = \{e_0, e_1, \ldots\}$; each edge e is *incident* to the elements of an unordered pair of vertices $\{u, v\}$ which are not necessarily distinct. These two vertices u and v are called the endpoints of the edge e. If the endpoints of an edge are not distinct, then the edge is called a *self loop*. Edges which have the same pair of endpoints are called *parallel*. In a finite graph both V and E are finite. In a non-directed graph the endpoints of an edge are unordered. A *path* is a sequence of edges $e_0, e_1, \ldots, e_n$ such that:

(1) e_i *and* e_{i+1} have a common endpoint, $0 \leq i < n$;
(2) If e_i is not a self loop, then it shares one of its endpoints with e_{i-1} and the other with e_{i+1} if it is not the first edge e_0 or the last edge e_n.

In a weighted graph each edge e is assigned a length $l(e)$. The length $l(P)$ of a path $P = e_0, e_1, \ldots, e_n$ is defined as:

$$l(P) = \sum_{i=0}^{n} l(e_i)$$

A path is called *simple* if no vertex appears on it more than once. A graph (V, E) is called *connected* if for every two vertices u and v, with $u, v \in V$, there exists a path whose start vertex is u and whose end vertex is v. A *circuit* is a path whose start

and end vertices are the same. A connected graph which has no circuits is called a *tree*.

For an example of a weighted graph let us consider fig. 2.1, consisting of the set of vertices {A, B, C, D, E, F } and the set of edges $\{e_0, e_1, ..., e_8\}$. Edges e_4 and e_5 are parallel. Edge e_8 is a self loop. The length of the simple path e_0, e_3, e_7 is 2 + 9 + 6 = 17. The path $e_0, e_1, e_2, e_7, e_6, e_4$ is a (simple) circuit. The graph in fig. 2.2 is a tree.

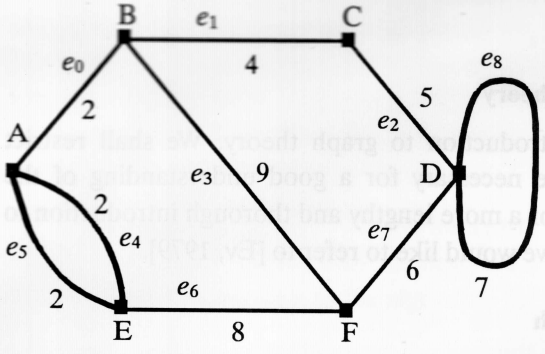


Fig. 2.1. A weighted non-directed graph

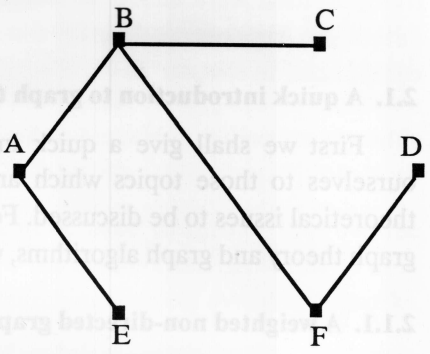


Fig. 2.2. A tree

2.1.2. A weighted directed graph

In a directed graph the endpoints of an edge are specifically ordered. The first endpoint of the edge is called the *start vertex* and the second the *end vertex*. The edge is said to be *directed* from its start vertex to its end vertex. Edges with the same start vertex and the same end vertex are called *parallel*, and if $u \neq v$ and $e_1 : u \to v$ and $e_2 : v \to u$ then e_1 and e_2 are *antiparallel*.

A directed path is a sequence of edges $e_0, e_1, ..., e_n$ such that the end vertex of e_i is the start vertex of e_{i+1}, $0 \leq i < n$. The length of a directed path and a simple directed path are defined similarly as in the undirected case. A directed graph (V, E) is called *strongly connected* if for every two vertices u and v, with u and $v \in V$, there exists a directed path from u to v; a directed path whose start vertex is u and whose end vertex is v. A weighted, directed, strongly connected graph is called a network.

2.2. Representing a railway network by a graph

A (physical) railway network can be represented by a weighted, non-directed, finite graph. The railway stations are represented by the vertices of the graph, the set V, and the connecting railroads by the edges, the set E. An edge e connects the vertices u and v, if and only if there exists a railroad connecting the stations

represented by these vertices. The length of an edge e, $l(e)$, is defined as the distance (in km, say) separating the stations connected by e. The length of a path in our railway network is the total distance covered by that path. For illustratory purposes, we consider a graph representing a small part of the Dutch railway network (simplified and modified); see fig. 2.3.

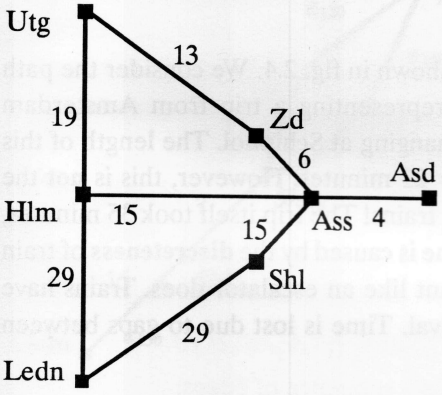


Fig. 2.3.

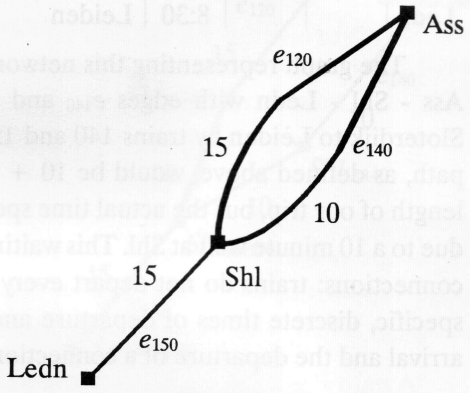


Fig. 2.4.

The vertices of the graph represent the stations (we use the official NS names and abbreviations[1]): Amsterdam Central Station (Asd), Amsterdam Sloterdijk (Ass), Haarlem (Hlm), Leiden (Ledn), Schiphol (Shl), Uitgeest (Utg) and Zaandam (Zd). The edges represent the rail sections Utg - Hlm, Utg - Zd, Hlm - Ledn, Hlm - Ass, Zd - Ass, Ass - Asd, Ass - Shl and Shl - Ledn. The lengths of the edges are the lengths of these sections as published in the NS distance tables for tariff calculation. If we consider the path Utg - Hlm - Ass - Shl - Ledn, then the length of this path is 78 km.

2.3. Representing a railway service network by a graph

A network of railway *services* can be represented by a weighted *directed* graph. Again the stations are represented by the vertices of the graph. An edge e directed from the start vertex u to the end vertex v represents a train running from the station represented by u (say A) to the one represented by v (say B). The length of an edge is the time this train takes to travel from A to B. Let us consider an example with

1 Sorry about Amsterdam Sloterdijk.

three stations (Ass, Shl and Ledn) and three trains: two from Ass to Shl (represented by the edges e_{120} and e_{140}) and one from Shl to Ledn (represented by the edge e_{150}).

	120	140	150	Station
Ass	7:40	7:55		Amsterdam Sloterdijk
Shl	7:55	8:05	8:15	Schiphol
Ledn			8:30	Leiden

The graph representing this network is shown in fig. 2.4. We consider the path Ass - Shl - Ledn with edges e_{140} and e_{150}, representing a trip from Amsterdam Sloterdijk to Leiden by trains 140 and 150, changing at Schiphol. The length of this path, as defined above, would be 10 + 15 = 25 minutes. However, this is **not** the length of our trip, but the actual time spent in trains! The trip itself took 35 minutes, due to a 10 minute wait at Shl. This waiting time is caused by the **discreteness** of train connections: trains do not depart every instant like an escalator does. Trains have specific, discrete times of departure and arrival. Time is lost due to gaps between arrival and the departure of a connection.

2.4. Representing waiting time

Since we would like the length of a path to be the same as the duration of the trip represented by the path, we have to find a way to include the waiting time in the length of the path. Some authors suggest to include a mean waiting time in the travel time (see for instance [Cl, 1972]). If a train goes, say every 10 minutes, then a 5 minute waiting time is added to the travel time. But even if the waiting time is chosen conservatively, this method just gives some estimate of how long a trip *may* be. It does not tell exactly when to leave, nor at what exact time we shall arrive. This is acceptable in stochastic applications (such as capacity planning or passenger flow models), where a mean travel time is sufficient; however, when the goal is to provide specific information about travelling possibilities, this method is not suitable.

Another way to represent the waiting time would be to create two vertices for each station. One vertex for arriving trains and one for departing trains. The two vertices are connected by an edge representing the waiting time. But then, what should be the length of this waiting edge? We cannot decide beforehand because the waiting time is dependent on the route: dependent on with which train we arrived and with which train we shall leave again. Let us consider our example (see fig. 2.5).

How long should we make the edge connecting Shl_{in} and Shl_{out}? If we would have arrived by train 120, the waiting time would be 20 minutes, if we would have arrived by train 140, 10 minutes. We can solve this problem by adding one vertex per

arriving train and one vertex per departing train. A vertex representing an arriving train at a station is connected to a vertex representing a departing train from that station by a waiting edge, if there is sufficient connection time. The length of the waiting edge corresponds to the appropriate waiting time. Our example would then look like fig. 2.6.

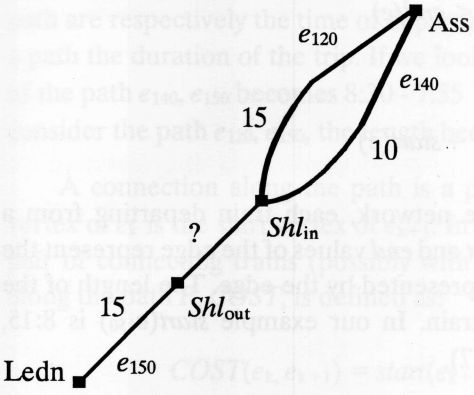


Fig. 2.5.

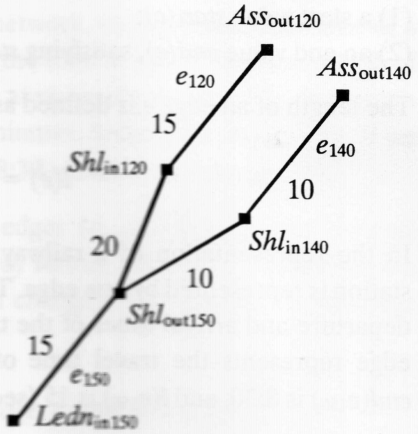


Fig. 2.6.

The above representation would lead to a very large graph in any practical situation. For instance, consider the Dutch railway service network, consisting of some 370 stations and almost 50 000 departures from these stations per day. The resulting graph would have 100 000 vertices (two vertices per train, one per departure of a train and one per arrival), and in the very best case 100 000 edges: 50 000 travelling edges representing journeys, and 50 000 waiting edges in the (improbable) case that we need only one waiting edge to connect an arriving train to the unique connecting departing train, clearly an unacceptable over-simplification. In the practical case of the Dutch railway service network, on average each station has about 140 arrivals and 140 departures. On average, each arriving train has a connection to 70 departing trains. So, per arrival we would need on average 70 *extra* edges representing a connection, giving $50\,000 * 70 = 3\,500\,000$ waiting edges in addition to the 50 000 travelling edges. Therefore, to represent the Dutch railway services network in this way, we need some 100 000 vertices and some 3 550 000 edges, while the network ignoring the connections could be represented by some 370 vertices and 50 000 edges.

2.5. A discrete network

To represent discrete connections in an efficient, adequate and more natural way, we propose a *discrete network*. In a discrete network the discreteness of the

connections is reflected in the properties of the edges. A discrete network consists of a finite, weighted, directed graph $G = (V, E)$. Moreover, with each edge e from E we associate two values:

(1) a start value *start(e)*;
(2) an end value *end(e)*, satisfying $start(e) < end(e)$.

The length of an edge e is defined as:

$$l(e) = end(e) - start(e) .$$

In the representation of a railway service network, each train departing from a station is represented by one edge. The *start* and *end* values of the edge represent the departure and arrival times of the train represented by the edge. The length of the edge represents the travel time of this train. In our example $start(e_{150})$ is 8:15, $end(e_{150})$ is 8:30, and $l(e_{150})$ is 15 (see fig. 2.7).

A path P in a discrete network is defined as a sequence of edges:

$$P = e_0, e_1, ..., e_n$$

such that

(1) The end vertex of e_k is the start vertex of e_{k+1}, $0 \leq k < n$;
(2) $end(e_k) \leq start(e_{k+1})$, $0 \leq k < n$

The start and end of a path P are defined as:

$$start(P) = start(e_0) \;\; ; \;\; end(P) = end(e_n).$$

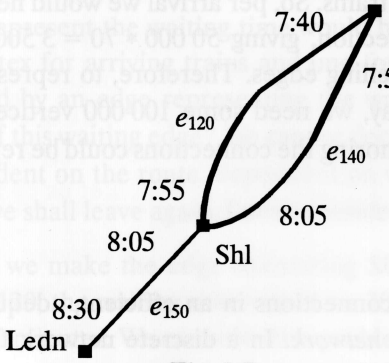


Fig. 2.7.

While the length of a path P, $l(P)$, is defined as:

$$l(P) = end(P) - start(P).$$

In the representation of a railway service network, the start and end value of a path are respectively the time of departure and the time of arrival, and the length of a path the duration of the trip. If we look again at our example (fig. 2.7), the length of the path e_{140}, e_{150} becomes 8:30 - 7:55 = 35 minutes. Indeed the travel time. If we consider the path e_{120}, e_{150}, the length becomes 8:30 - 7:40 = 50 minutes.

A connection along the path is a pair of edges $\{e_k, e_{k+1}\}$, such that the end vertex of e_k is the start vertex of e_{k+1}. In a railway service network a connection is a pair of connecting trains (possibly with a train change). The cost of a connection along the path P, $COST$, is defined as:

$$COST(e_k, e_{k+1}) = start(e_{k+1}) - end(e_k), \text{ with } 0 \leq k < n.$$

In the representation of a railway service network, the $COST$ of a connection represents the time one has to wait for the next train at a station.

3. Searching A Graph

3.1. Shortest path algorithms

For finding the best railway connections between two stations we need an algorithm which searches for the shortest path in the graph representing the railway service network. We distinguish two basic types of graph search algorithms:

(1) Matrix algorithms.
(2) Tree building algorithms.

3.1.1. Matrix algorithms: Floyd's algorithm

Matrix algorithms (see for instance [Da, 1966], [Fl, 1962], [Hu, 1967], [Ye, 1968]) compute the shortests distances and paths between all vertices of the graph simultaneously by manipulating a $|V| * |V|$ matrix. Since we need the shortest path between two specific vertices only, the source vertex and the goal vertex, these algorithms are inefficient for our purpose. However there is one interesting property of the matrix algorithm by Floyd [Fl, 1962]: it allows negative length edges and even negative length circuits in a directed, weighted graph. We shall describe Floyd's algorithm.

Let G be a finite directed graph (V, E). $V = \{1, 2, .., n\}$. Each edge e from E has a length $l(e)$, which may be negative. Define an n by n matrix δ^0 in which:

$$\delta^0(i,j) = l(e), \quad \text{if} \quad e: i \rightarrow j, \; e \in E$$

and

$$\delta^0(i,j) = \infty, \quad \text{otherwise.}$$

Floyd's algorithm is as follows:

(1) $k \leftarrow 1$.
(2) For every $1 \leq i,j \leq n$ compute $\delta^k(i,j) \leftarrow$ Min $\{ \delta^{k-1}(i,j), \delta^{k-1}(i,k) + \delta^{k-1}(k,j)\}$.
(3) If $k = n$ then stop.
Else $k \leftarrow k + 1$ and go to step (2).

It can be shown that after termination, $\delta^n(i,j)$ contains the length of the shortest path from i to j. See for instance, [St, 1974] or [Ev, 1979]. In the kth step of Floyd's algorithm the matrix contains all shortest paths, allowing to use only a subset of vertices 1 to k as intermediate vertices. The basic operation is to check in the kth step whether a route can be improved by using a route already found (which was found allowing only vertices 1 through k-1 as an intermediate vertex), and going through vertex k (thus a route allowing vertices 1 through k as intermediate vertices). Notice that during each phase of the algorithm all matrix entries are tried to construct better paths. This means that even non-existent connections (characterized by a matrix entry containing ∞) are tried. At all times, only (the length of) the best known path between each possible pair of vertices i and j is stored in the matrix entry $\delta(i,j)$.

3.1.2. Tree building algorithms

Tree building algorithms, also known as labeling algorithms, build a tree of paths from the source to the other vertices of the graph. Generally, the shortest paths from one vertex to all other vertices are found. In these algorithms, the distance of the currently best know path from the starting vertex to a vertex v is remembered by its label $\lambda(v)$. All tree building algorithms use the same principle: first they initialize all distances (λ's) to infinity, during the remaining of the searching process the characteristic operation is:

$$\lambda(v) = \text{Min}(\lambda(v), \lambda(u) + l(e_{u \to v}))$$

This operation checks whether the currently known shortest distance between the source vertex and vertex v (which is the label of v, $\lambda(v)$), can be reduced by adding the edge $e_{u \to v}$, with length $l(e_{u \to v})$, to the shortest path tree. It is checked whether the current distance to v can be reduced by using a path via vertex u. Notice that only existing connections are tried (which was not the case in Floyd's matrix algorithm).

Within the tree building algorithms two types of algorithms are distinguished:

(1) Label correcting algorithms;
(2) Label setting algorithms.

During the search process of a tree building algorithms three types of vertices can be distinguished, shown in figure 3.1.

(1) vertices that are part of the current shortest path tree;
(2) vertices that are adjacent to the vertices in the shortest path tree;
(3) other vertices.

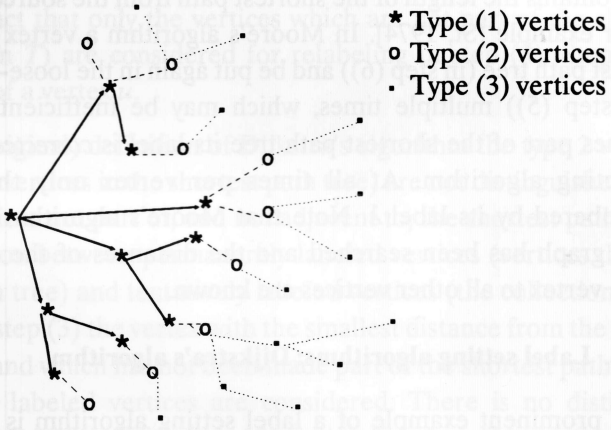


Fig. 3.1.

The vertices adjacent to the vertices in the shortest path tree are also called *loose-end* vertices, as they are adjacent to the loose ends of the tree branches.

3.1.2.1. Label correcting algorithms: Moore's algorithm

A prominent example of a label correcting algorithm is an algorithm known as Moore's algorithm [Mo, 1957], similar algorithms were published by Ford [Fo, 1956] and Bellman [Be, 1958]. In his paper Moore describes four algorithms, we shall limit ourselves to the most important one. The others are, in fact, derived special cases.

Consider a finite directed graph *(V, E)*. Each edge e from E has a length $l(e)$ which may be negative. The two special vertices s and t of the graph are the *starting* and *terminating* vertices. We want to find a shortest directed path from s to t, where the length of a path is the sum of the lengths of its edges. The label $\lambda(v)$ of a vertex v was explained above. Here is Moore's algorithm:

(1) $\lambda(s) \leftarrow 0$ and for all $v \in V, v \neq s$, $\lambda(v) \leftarrow \infty$.
(2) $T \leftarrow \{s\}$.
(3) If T is empty, stop.

(4) Let u be a vertex in T.
(5) For every edge $e : u \rightarrow v$,
    if $\lambda(v) > \lambda(u) + l(e)$
    then $\lambda(v) \leftarrow \lambda(u) + l(e)$ and $T \leftarrow T + \{v\}$.
(6) $T \leftarrow T - \{u\}$ and go to step (3).

It can be shown that after termination in step (3), the label of each vertex u, $\lambda(u)$, contains the length of the shortest path from the source vertex s to the vertex u, see for example [St, 1974]. In Moore's algorithm a vertex can become part of the shortest path tree (in step (6)) and be put again in the loose-end table (the collection T, in step (5)) multiple times, which may be inefficient. Each time the vertex becomes part of the shortest path tree its label is *corrected*. Hence the term label correcting algorithm. At all times per vertex only the best known path is remembered by its label λ. Note that Moore's algorithm does not stop until the entire graph has been searched and the distances of the shortest paths from the source vertex to all other vertices are known.

3.1.2.2. Label setting algorithms: Dijkstra's algorithm

A prominent example of a label setting algorithm is an algorithm known as Dijkstra's algorithm [Di, 1959]. Similar algorithms were published by Whiting and Hillier [Wh, 1960]. We shall describe Dijkstra's algorithm, which is often considered to be the best algorithm to search a finite, directed graph whose edges have non-negative lengths.

Consider a finite directed graph (V, E), where V is the set of vertices and E the set of directed edges joining two vertices from V. Each edge e from E has a length $l(e) \geq 0$. The two special vertices s and t of the graph are the *starting* and *terminating* vertices. We want to find a shortest directed path from s to t, where the length of a path is the sum of the lengths of its edges. The label $\lambda(v)$ of a vertex v was explained above.

(1) $\lambda(s) \leftarrow 0$ and for all $v \in V, v \neq s$, $\lambda(v) \leftarrow \infty$.
(2) $T \leftarrow V$.
(3) Let u be a vertex in T for which $\lambda(u)$ is minimum.
(4) If $u = t$, stop.
(5) For every edge $e : u \rightarrow v$,
    if $v \in T$ and $\lambda(v) > \lambda(u) + l(e)$
    then $\lambda(v) \leftarrow \lambda(u) + l(e)$.
(6) $T \leftarrow T - \{u\}$ and go to step (3).

Dijkstra's algorithm can be shown to find an optimal path (when available) from s to t, see for example [Ev, 1979]. Each vertex in the network is labeled with its distance λ from the source vertex along the best path that is known at the time of labeling. When the label of a vertex is made *permanent* in step (3), we have found a (there may be ties) shortest route (from s) to that vertex. The difference with Moore's algorithm lies in the selection of the vertex to be *examined* from the loose-end vertices (step (4) in Moore's algorithm and step (3) in Dijkstra's algorithm) and in the fact that only the vertices which are not part of the shortest path tree (the collection T) are considered for relabeling in step (5). Step (5) is called the examination of a vertex u.

Note that in this (original) definition of Dijkstra's algorithm the type 2 vertices (vertices adjacent to the vertices in the shortest path tree) are not distinguished from the type 3 vertices (vertices not part of, and not adjacent to, the shortest path tree). There is only a distinction between permanently labeled vertices (vertices that are part of the shortest path tree) and tentatively labeled vertices (the collection T). In Dijkstra's algorithm, in step (3) the vertex with the smallest distance from the source (i.e. the smallest label) and which has not been made part of the shortest path tree is chosen. All tentatively labeled vertices are considered. There is no distinction between vertices labeled with infinity (type 3 vertices) and vertices with a label unequal infinity (type 2 vertices). By choosing the vertex with the smallest label, no shorter path to this vertex can possibly be found later on. When a vertex becomes part of the shortest path tree (in step (6)), its label λ is made *permanent*. It is set and cannot change anymore. Hence the term label setting algorithm. Also, a vertex is examined only once. That is why these algorithms are also called *once-through* algorithms (the term was suggested by Murchland [Mu, 1967]). Furthermore, Dijkstra's algorithm stops as soon as the length of the shortest path from the source to the goal vertex is known. This may mean that for other vertices than the goal vertex, the length of the shortest path from the source has not yet been determined. By changing the stopping criterion in step (4) to a test whether T is empty, the algorithm will not stop until all shortest paths have been found. Clearly these properties make Dijkstra's algorithm, and label setting algorithms in general, more efficient than label correcting algorithms. Note that in a label setting algorithm also, at all times, per vertex only (the length of) the best known path is remembered by its label λ.

3.1.2.3. An improvement of Dijkstra's algorithm

As mentioned above, in the original definition of Dijkstra's algorithm all tentatively labeled vertices are considered for examination in step (3). No distinction

is made between vertices labeled with infinity (type 3 vertices) and vertices with a label unequal infinity (type 2 vertices), as with Moore's algorithm. We can easily incorporate such a distinction in Dijkstra's algorithm by the introduction of a loose-end table: a collection F which we prefer to call the *frontier*. Dijkstra's algorithm becomes:

(1) $\lambda(s) \leftarrow 0$ and for all $v \in V, v \neq s$, $\lambda(v) \leftarrow \infty$.
(2) $T \leftarrow V$, $F \leftarrow \{s\}$.
(3) Let u be a vertex in F for which $\lambda(u)$ is minimum; if F is empty then stop, no path could be found.
(4) If $u = t$, stop, an optimal path is found.
(5) For every edge $e: u \rightarrow v$,
    if $v \in T$ and $\lambda(v) > \lambda(u) + l(e)$
    then $\lambda(v) \leftarrow \lambda(u) + l(e)$ and $F \leftarrow F + \{v\}$.
(6) $T \leftarrow T - \{u\}$, $F \leftarrow F - \{u\}$ and go to step (3).

As before, the collection V contains all vertices which have not been made permanent yet. The collection F contains all vertices which have not been made permanent yet, but which have been visited at least once. Selecting the vertex to be examined, it makes no sense to consider vertices which have never been visited yet and which are still labeled with infinity. For when we make a vertex permanent, its label is the length of the shortest path from the source to this vertex, and in order to know that distance, we must have visited this vertex at least once. Considering only the vertices in the frontier makes the algorithm more efficient, especially in the early stages of the searching process when most vertices have never been visited yet and are still labeled with infinity.

3.2. Remembering the route of the shortest path

The algorithms as discussed above determine the length of the shortest path between two (or more) vertices. However, usually we are not only interested in the length of the shortest path, but also in the route of the shortest path. In order to obtain the route of the shortest path in the algorithms discussed above, apart from a label, a vertex is given a *backvertex*. The backvertex is set to the vertex from which the current label was set (the vertex u in step (5) of the description of Dijkstra's algorithm). By traversing the backvertices the route of the shortest path can be constructed.

3.3. The dynamic programming principle

The property that characterizes all shortest path algorithms is that at all times per vertex only the best known path is remembered. The foundation of this property is that when we have found a shortest route from s to t, if this shortest route passes through vertex b, then a shortest route from s to b, a partial path, is part of the shortest route from s to t (the complete path). There may be several shortest paths; this slight complication will be ignored here. If we remember the best known path per vertex, we can never miss an optimal path in case this vertex turns out to be on an optimal path. Therefore, for each vertex only *one* path, the best partial path, needs to be remembered. This result is known as the *Markovian property*, the *principle of optimality* for dynamic programming (see, for instance [Hi, 1986]) and as the *dynamic programming principle* (see for instance [Wi, 1984]). For problems in general, the Markovian property means that knowledge of the current state conveys all the information about its previous behaviour necessary for determining the optimal policy henceforth. Or more theoretically: the conditional probability of any future "event", given any past "event" and the present state, is independent of the past event and depends upon only the present state of the process.

3.4. Searching bidirectionally

When only the shortest path between two vertices is needed, it is appealing to start searching from both the source vertex and the goal vertex simultaneously. According to Murchland [Mu, 1967], when using a bidirectional algorithm, the savings in computation time over the normal, unidirectional algorithm is about 50 percent. A modified label setting algorithm can be used to construct trees from both the source and the goal, adding branches to the two trees in an alternating way. Whenever the two trees touch we have found a path from the source to the goal. The important issue is how to decide when the algorithm may be terminated, preserving optimality of solution. We cannot stop after the first time we have found a complete path. The stopping criterion was given by Nicholson [Ni, 1966].

Suppose that the last vertex we have made permanent in the tree built from the source vertex is u_s, and that u_g is the last vertex made permanent in the tree built from the goal vertex. Then we know that all vertices adjacent to vertices of the shortest path tree from the source vertex have been visited, and moreover that they have a distance from the source greater than or equal to $\lambda(u_s)$. Similarly, all vertices adjacent to vertices of the shortest path tree from the goal vertex have been visited and they have a distance from the goal greater than or equal to $\lambda(u_g)$. So, every path from the source to the goal vertex using a vertex which is not part of either shortest path tree, must have a length of at least $\lambda(u_s) + \lambda(u_g)$. It is easily seen that if the

length of a shortest path from the source to the goal vertex which has actually been found (this path must use only vertices which are part of at least one of the shortest path trees, otherwise we would not have found it yet) is smaller than or equal to $\lambda(u_s) + \lambda(u_g)$, that this shortest path is an optimal path. For more discussion of bidirectional search see [Lu, 1989] and [Po, 1971].

3.5. Conclusion

Since we need a graph search algorithm giving a shortest path between two vertices only, a matrix algorithm which searches for shortest paths between all vertices simultaneously, is inefficient for our purpose. Of the tree building algorithms, which search for the shortest path from one vertex to all other vertices, the label setting type is the most efficient one. So for our application, the obvious choice is to use (the improved) Dijkstra's algorithm as described above. We shall not use a bidirectional version for reasons which will become clear later on.

4. Searching Discrete Networks

4.1. Adapting Dijkstra's algorithm to discrete networks

Now that we have decided which graph search algorithm is best suited for our purpose, Dijkstra's algorithm, we adapt it to discrete networks. In a discrete network each edge has a discrete start and end value, so in Dijkstra's algorithm we have to change the labeling step (step (5)). In Dijkstra's algorithm we add the length of an edge to the label of the start vertex of this edge and compare it to the label of the end vertex. In a discrete network we compare the end value of an edge to the label of its end vertex. Furthermore, before trying an edge, we must check whether its start value is greater than or equal to the label of the start vertex (the condition of a discrete path). Finally we must also give a desired start value of the path (which would correspond to the desired time of departure in a railway service network representation). Here is Dijkstra's algorithm for a discrete network (we use our improved version):

Consider a discrete network (V, E), where V is the set of vertices and E the set of directed edges. Each edge e from E has a start value $start(e)$ and an end value $end(e)$, with $start(e) < end(e)$. The two special vertices s and t of the network are the *starting* and *terminating* vertices. We want to find a discrete path from s to t in our discrete network, where the end value of the path is minimum and the start value of the path is at least T_{start}.

(1) $\lambda(s) \leftarrow T_{start}$ and for all $v \in V, v \neq s$, $\lambda(v) \leftarrow \infty$.
(2) $T \leftarrow V$, $F \leftarrow \{s\}$.
(3) Let u be a vertex in F for which $\lambda(u)$ is minimum; if F is empty then stop, no path could be found.
(4) If $u = t$, stop, an optimal path is found.
(5) For every edge $e: u \rightarrow v$ for which $start(e) \geq \lambda(u)$,
if $v \in T$ and $\lambda(v) > end(e)$
then $\lambda(v) \leftarrow end(e)$ and $F \leftarrow F + \{v\}$.
(6) $T \leftarrow T - \{u\}$, $F \leftarrow F - \{u\}$ and go to step (3).

We proof by induction on the number of steps that when a vertex u becomes permanent in step (3), we have found a path from s to u for which the end value is minimum. In the first step, we know that $u = s$. Since $\lambda(s)$ is T_{start} this is evidently true. Suppose at the kth step we make u_n permanent with a value of T_k determined by some path $P_1 = s, e_0, u_1, ..., u_{n-1}, e_{n-1}, u_n$. Now suppose there exists another path from s to u_n with an end value less than T_k, say the path $P_2 = s, e_a, v_{a+1}, ..., v_p, e_p, u_n$. There are two possibilities: either vertex v_p has already been made permanent, or it has not been. If it has, then e_p must already have been tried when v_p was made permanent, so u_n would have been labeled then and we would have found that path. Now consider the case that v_p is still tentatively labeled. Either the label of v_p is greater than or equal to the label of u_n, or it is smaller. In the first case the path P_2 cannot have a smaller end value than T_k since the end value of the edge e_p connecting v_p and u_n, must be greater than its start value, which must be at least the label of v_p. If the label of v_p is smaller than the label of u_n, then v_p would have become permanent at the k_{th} instead of u_n, allowing u_n to be labeled from v_p (using e_p).

4.2. The principle of optimality for discrete networks

The foundation of the algorithm is formed by an assumption similar to the dynamic programming principle. We shall refer to this assumption as the principle of optimality for discrete networks:

In a discrete network, let $\lambda(u)$ denote the minimum end value of any path from s to u in the network. If there exists a path in the network, from s to t which passes through vertex u and with a minimum end value, then the path which gave u its label $\lambda(u)$ is part of a path from s to t with a minimum end value.

For the proof, suppose the contrary: there exists an optimal (discrete) path from s to t which passes through u:

$$P_{opt} = s, e_0, v_1, ..., e_j, u, e_{j+1}, ..., e_q, t$$

with $end(e_j) \geq \lambda(u)$, and suppose there does not exist an optimal path for which the end value of the partial path to u is $\lambda(u)$.

We know that

$$end(e_j) \geq \lambda(u).$$

Suppose that the partial path which gave u its label λ is:

$$P_\lambda = s, e_a, v_{a+1},..., e_k, u.$$

Since by the definition of λ, $end(e_k) = \lambda(u)$, we know that

$$end(e_j) \geq end(e_k).$$

And since $start(e_{j+1})$ must be at least $end(e_j)$, we know that

$$start(e_{j+1}) \geq end(e_k).$$

But then it is possible to construct an optimal path from s to t:

$$s, e_a, v_{a+1},..., e_k, u, e_{j+1},..., e_q, t$$

a path which passes trough u, for which the partial path to u has an end value of $\lambda(u)$. Which is a contradiction.

4.3. An example

Let us consider the following example (see fig. 4.1). Suppose we want to travel from Utg to Asd, and that we want to depart at or after 7:00.

	100	**110**	**160**	**Station**
Utg	7:00	7:10		Uitgeest
Hlm	7:15	7:25		Haarlem
Ass	7:30	7:40	7:45	Amsterdam Sloterdijk
Asd			7:50	Amsterdam Central Station

In the first step, Utg gets labeled with 7:00 and is put in the frontier. Utg becomes permanent and all outgoing edges from Utg are tried, in this case edges 100 and 110. Hlm is labeled 7:15 by trying edge 100, and is not relabeled by edge 110. Hlm becomes permanent and from Hlm we try edges 100 and 110. Ass gets labeled 7:30 by 100 and is not relabeled by edge 110. Ass becomes permanent and edge 160 is used to label Asd 7:50. Then Asd becomes permanent and we have found a path with an optimal end value.

4.4. Relevant edges

The first thing that can be noticed from the previous example is that we try edges which are irrelevant. When Utg became permanent, we tried edge 100 and then edge 110. However, since edge 100 could be applied, and since edge 110 has the

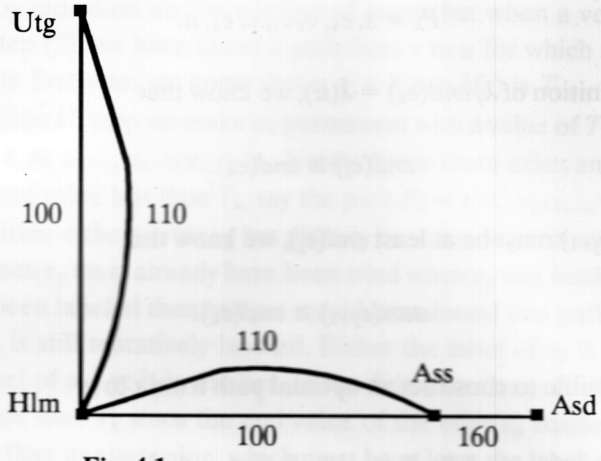


Fig. 4.1.

same end vertex and the same length as edge 100, but a higher start value, we know for sure that edge 110 can never be used to improve a label. Given a label and an end vertex, only one edge departing from a vertex needs to be tried: the *relevant* edge. We define the relevant edge as follows:

Given a vertex u and a vertex v, then the relevant edge from u to v is the edge $e: u \rightarrow v$ for which the following two *ordered* conditions hold:

(1) $start(e) \geq \lambda(u)$,
(2) $end(e)$ is minimum.

With this definition, step (5) of our algorithm can be changed to:

(5) For every relevant edge $e: u \rightarrow v$,
if $v \in T$ and $\lambda(v) > end(e)$
then $\lambda(v) \leftarrow end(e)$ and $F \leftarrow F + \{v\}$.

This way, only one edge per neighbour is tried for relabeling this neighbour. If the relevant edge can be used to relabel the neighbour, than the other edges cannot be used to improve the label even further since for any other edge to this neighbour but the relevant edge, either its start value is smaller than the label of the start vertex (so it may not be used to construct a discrete path), or its end value is greater than or equal to the end value of the relevant edge, so it cannot further improve the label. If the relevant edge cannot be used to relabel a neighbour, then neither can any of the other edges to this neighbour for the same reasons.

4.5. Suboptimality of solution

The second thing that can be seen from the previous example is that, although the algorithm does give us a path with an optimal end value, this is not really the solution we would like to get. We asked for a trip from Utg to Asd departing at or after 7:00. The proposed solution to take train 100 from Utg to Ass and train 160 from Ass to Asd, indeed makes us arrive as early as possible (7:50) while departing at 7:00. So, this answer does satisfy our question. But we could have made another trip also arriving at 7:50, but departing 10 minutes later by taking train 110 from Utg to Ass, instead of train 100. Obviously this solution is better! This problem is caused by the discreteness of the connections. Or more to the point, by the cost of the connections: *COST* (see chapter 2). In the example, since we had to wait for 15 minutes for our next train at Ass, we might have taken one later train from Utg to Ass, still arriving in time for our connection.

4.6. An optimal path in a discrete network

Clearly, we need a better definition of what we consider to be an optimal path in a discrete network. In a railway service network, an optimal trip makes us arrive as early as possible, and given this earliest arrival time, allows us to leave as late as possible, making the trip as short as possible. Similarly we define an optimal path in our discrete network. Given a discrete network $G = (V, E)$, two vertices $s, t, \in V$, and a starting value T_{start}, a discrete path P from s to t in G is optimal if the following three *ordered* conditions hold:

(1) $start(P) \geq T_{start}$, and
(2) $end(P)$ is minimum, and
(3) $start(P)$ is maximum, or equivalently $l(P)$ is minimum.

If a path P satisfies the first two conditions only, we call P suboptimal.

4.7. Traversing the suboptimal solution

From our example it may seem that, in order to find the optimal solution given a suboptimal solution, it suffices to traverse the suboptimal solution in a backward fashion, trying to improve it. At each vertex, we search for the edge from its backvertex, with the highest possible end value smaller than or equal to the label of the current vertex (the end vertex of the edge). In our example, at Ass we search for the edge connecting Hlm and Ass with the highest end value smaller than or equal to 7:45. This way the optimal solution would be found. In general, however, there is no guarantee that the optimal solution always uses the same route as the suboptimal solution. For instance, consider the following example (see fig. 4.2).

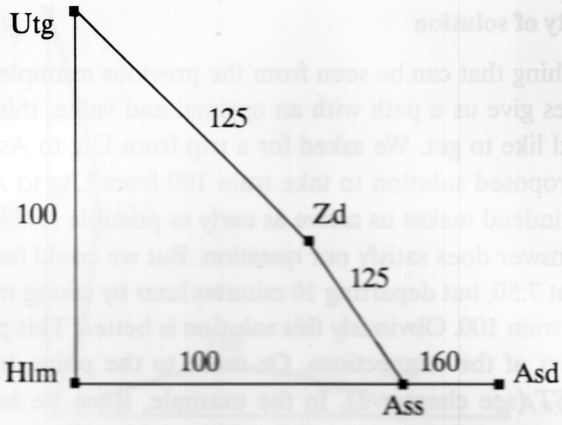


Fig. 4.2.

	100	125	160	Station
Utg	7:00	7:10		Uitgeest
Hlm	7:15	|		Haarlem
Zd	|	7:30		Zaandam
Ass	7:30	7:40	7:45	Amsterdam Sloterdijk
Asd			7:50	Amsterdam Central Station

For a trip from Utg to Asd leaving at 7:00, the suboptimal solution found by our algorithm is by the route Utg, Hlm, Ass, Asd (train 100 and train 160, departing at 7:00 and arriving at 7:50) while the optimal solution is by the route Utg, Zd, Ass, Asd (train 125 and train 160, departing at 7:10 and arriving at 7:50). This optimal solution cannot be found by simply traversing the suboptimal solution.

We are not able to find the optimal solution with our algorithm because a non-minimum label might lead to an optimal solution. The smallest, minimum label may, or may not. It all depends on what the start value of the next connection will be, which in turn depends on its next connection etc., ultimately depending on the start value of the final edge, and thus its end value. In the end the greatest possible start value depends on the smallest possible end value of the path. If we would want to be able to find the optimal solution in our algorithm, instead of having to develop further only one arriving path per vertex (the one giving it its minimum label), we would have to develop further all arriving paths (ending with non-minimum labels), resulting in a combinatorial explosion.

4.8. Finding the optimal solution: the second pass

To avoid a combinatorial explosion in the first pass, in order to find the optimal solution we make a second pass. After we have found the earliest possible end value

of a path, we conduct a second, backward search in order to find the matching earliest possible start value. This way the first pass can remain the same, avoiding a combinatorial explosion. For the backward search Dijkstra's algorithm is changed in a straightforward way. Let the label of vertex in the second, backward search be denoted by κ. The algorithm for the backward pass then becomes:

(1) $\kappa(t) \leftarrow \lambda(t)$ and for all $v \in V, v \neq t$, $\kappa(v) \leftarrow -\infty$.
(2) $T \leftarrow V$, $F \leftarrow \{t\}$.
(3) Let u be a vertex in F for which $\kappa(u)$ is maximum; if F is empty then stop, no path could be found.
(4) If $u = s$, stop, an optimal path is found.
(5) For every relevant edge $e: v \rightarrow u$,
if $v \in T$ and $\kappa(v) < start(e)$
then $\kappa(v) \leftarrow start(e)$ and $F \leftarrow F + \{v\}$.
(6) $T \leftarrow T - \{u\}$, $F \leftarrow F - \{u\}$ and go to step (3).

Note that we are searching backwards: we search from the terminating vertex t to the starting vertex s. At the start the label of t is made the value that we have found in the forward search as the smallest possible end value, which is the label $\lambda(t)$ of the goal vertex. Since we are searching backwards, we search for a path with a *maximum* start value. Also note that for the backward search the definition of a relevant edge is different. In the backward search the relevant edge from u to v is the edge $e: u \rightarrow v$ for which the following two *ordered* conditions hold:

(1) $end(e) \leq \kappa(v)$,
(2) $start(e)$ is maximum.

The proof of the backward version of the algorithm is similar to the proof of the forward version.

4.9. Using the results from the first search

Our second, backward search need not be a complete search. We can use the results from the first pass to limit the search of the second pass. In the previous example, when traversing the optimal solution, we would have wanted to know that we should go back not only to Hlm, but also to Zd.

First, in the backward search, we only need to consider those vertices u for which a minimum path was found in step (3) of the (forward) algorithm. The other vertices in the network could not be reached by a path with an end value smaller than $\lambda(t)$ (since they had not become permanent yet). Therefore these vertices can never

be on a path arriving at t with an end value smaller than or equal to $\lambda(t)$, and with a start value of at least T_{start}.

For the proof, suppose that vertex u was not made permanent in the first, forward search, but there does exist a path $s,..., u,..., t$ for which the start value is at least T_{start} and the end value smaller than or equal $\lambda(t)$. Then obviously, $\lambda(u) < \lambda(t)$ (since we do not allow zero-length edges in a discrete network), so in the forward search u would have been made permanent before t. If we would allow zero-length edges, by the way, we would have to make sure that, if in step (3) of the algorithm there are multiple vertices with a minimum label, then a non-terminal vertex is chosen.

Second, the additional information we want (i.e. also try Zd in the example) becomes available if we store *neighbour dependent* λ's, i.e. the smallest end value that could be used to label a vertex from each of its neighbours, instead of storing the best end value only. Per neighbour we store which was the smallest end value which was tried for relabeling. Even if the relabeling did not actually take place because the vertex had been labeled with a smaller value from another neighbour already! In our second, backward search, we only need to consider those neighbours v of a vertex u, for which its λ would have been at most $\kappa(u)$. Let us denote this neighbour dependent λ by ω. So, the smallest end value that was tried to label u from v is $\omega(u, v)$.

For the proof, note that from the correctness of our forward pass we know that there does not exist a path $s,..., v, u$ for which the start value is at least T_{start} and for which the end value is smaller than $\omega(u, v)$. Similarly, we know that when we make u permanent in our backward search, there does not exist a path $u,..., t$ for which the start value is smaller than $\kappa(u)$, and the end value is at most $\lambda(t)$ (the end value of the suboptimal path for which we have to determine the matching maximum start value). So, if $\omega(u, v) > \kappa(u)$, then there cannot exist a path $s,..., v, u,..., t$ for which both the start value is at least T_{start}, and the end value is at most $\lambda(t)$.

In our example, ω(Ass, Hlm) is 7:30 and ω(Ass, Zd) is 7:40. Since κ(Ass) is 7:45 we need to visit both Hlm and Ass. Now consider the next example (see fig. 4.3):

	100	**160**	**135**	**Station**
Utg	7:00		7:20	Uitgeest
Hlm	7:15		|	Haarlem
Zd	|		7:40	Zaandam
Ass	7:30	7:45	7:50	Amsterdam Sloterdijk
Asd		7:50		Amsterdam Central Station

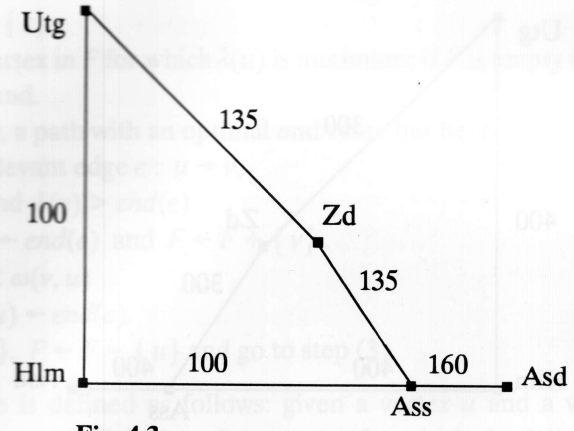


Fig. 4.3.

In this example ω(Ass, Hlm) is again 7:30 and ω(Ass, Zd) is 7:50. Since κ(Ass) is 7:45 we do have to visit Hlm but we do not have to visit Zd. In our first, forward search we determined that Ass could not be reached earlier than 7:50 travelling by Zd, departing at or after 7:00 (which is stored by ω(Ass, Zd)). In our second, backward search we find that if we want to arrive at Asd at 7:50, we must be able to arrive at Ass before or at 7:45 (which is stored by κ(Ass)). So, in our backward search it is not necessary to visit Zd, for it is not possible to travel via Zd if we want to depart at or after 7:00 and arrive at 7:50.

Earlier we said that it is important to record the smallest possible label from each neighbour even if relabeling does not actually take place. Let us take a look at another example to see why (see fig. 4.4).

	300	**400**	**Station**
Utg	7:00	7:20	Uitgeest
Hlm	|	7:35	Haarlem
Zd	7:20	|	Zaandam
Ass	7:30	7:50	Amsterdam Sloterdijk
Asd		7:55	Amsterdam Central Station

Suppose we want to travel from Utg to Asd, departing at or after 7:00. In the forward search, first Utg becomes permanent labeled 7:00. Zd becomes permanent labeled 7:20, Ass becomes permanent labeled 7:30 and Hlm becomes permanent labeled 7:35. When the neighbours of Hlm are examined, Ass is not even tried for relabeling because Ass is not a member of T, as is required in step (5). Ass is already part of the minimum path tree, so its label cannot be further improved. Because Ass is not even tried for relabeling, ω(Ass, Hlm) is **not** set to 7:50. After we have found

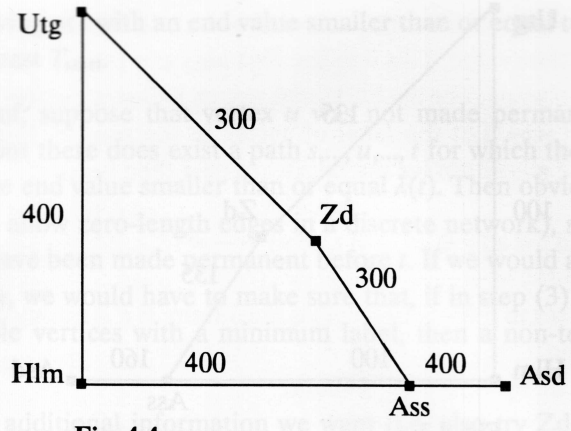


Fig. 4.4.

the path Utg, Zd, Ass, Asd representing a trip with trains 300 and 400, we start searching backwards. In the backward search, κ(Ass) becomes 7:50 but since we did not record ω(Ass, Hlm) we do not visit Hlm.

In order to ensure optimality, we must make a record of the fact that we *could* have tried to relabel a vertex with a certain value, even though we did not even try to relabel it because the vertex had already been made permanent. We invest a little in the forward search in order to gain in the backward search. This way, in the backward search the search space examined can be significantly smaller than a full backward search.

4.9.1. A two-pass algorithm: DISNET

Finally, we give a formal definition of the two-pass search algorithm which finds an optimal path in a discrete network (if there exists one). We shall refer to this algorithm as the *DISNET* algorithm.

Consider a discrete network (V, E), where V is the set of vertices and E the set of directed edges. Each edge e from E has a start value *start(e)* and an end value *end(e)*, with *start*(e) < *end*(e). The two special vertices s and t of the network are the *starting* and *terminating* vertices. We want to find a discrete path from s to t in our discrete network, where the end value of the path is minimum and given this end value, the start value of the path is maximum and at least T_{start}.

Pass 1:

(1) $\lambda(s) \leftarrow T_{\text{start}}$ and for all $v \in V$, $v \neq s$, $\lambda(v) \leftarrow \infty$.
For all $v \in V$, $\omega(v, u) = \infty$ for every neighbour u of v.

(2) $T \leftarrow V$, $F \leftarrow \{s\}$.
(3) Let u be a vertex in F for which $\lambda(u)$ is minimum; if F is empty then stop, no path could be found.
(4) If $u = t$, stop, a path with an optimal end value has been found.
(5) For every relevant edge $e: u \rightarrow v$,
if $v \in T$ and $\lambda(v) > end(e)$
then $\lambda(v) \leftarrow end(e)$ and $F \leftarrow F + \{v\}$.
if $end(e) < \omega(v, u)$
then $\omega(v, u) \leftarrow end(e)$.
(6) $T \leftarrow T - \{u\}$, $F \leftarrow F - \{u\}$ and go to step (3).

A relevant edge is defined as follows: given a vertex u and a vertex v, then the relevant edge from u to v is the edge $e: u \rightarrow v$ for which the following two *ordered* conditions hold:

(1) $start(e) \geq \lambda(u)$, and
(2) $end(e)$ is minimum.

Pass 2:

(1) $\kappa(t) \leftarrow \lambda(t)$ and for all $v \in V, v \neq t$, $\kappa(v) \leftarrow -\infty$.
(2) $T \leftarrow V$, $F \leftarrow \{t\}$.
(3) Let u be a vertex in F for which $\kappa(u)$ is maximum.
(4) If $u = s$, stop, an optimal path is found.
(5) For every relevant edge $e: v \rightarrow u$, with $\omega(u, v) \leq \kappa(v)$,
if $v \in T$ and $\kappa(v) < start(e)$
then $\kappa(v) \leftarrow start(e)$ and $F \leftarrow F + \{v\}$.
(6) $T \leftarrow T - \{u\}$, $F \leftarrow F - \{u\}$ and go to step (3).

A relevant edge is defined as follows: given a vertex u and a vertex v, then the relevant edge from u to v is the edge $e: u \rightarrow v$ for which the following two *ordered* conditions hold:

(1) $end(e) \leq \kappa(v)$, and
(2) $start(e)$ is maximum.

By the correctness of the first pass, after the first pass we have found a path P_1 for which $start(P_1) \geq T_{start}$ and for which $end(P_1)$ is minimum. By the correctness of the second pass, we know that after the second pass we have found a path P_2 for which in addition, $start(P_2)$ is maximum. So, the algorithm gives an optimal solution.

5. Discrete Dynamic Networks

5.1. Visiting costs

Until now we have been ignoring one particular property of travelling by train: time is required to change trains. In a graph representation this translates into a visiting cost at a vertex. In the previous chapter, a train change was dealt with in the graph, by the definition of a relevant edge. For instance, in the forward pass of the algorithm, the definition of a relevant edge is: given a vertex u and a vertex v, then the relevant edge from u to v is the edge e: $u \rightarrow v$ for which the two following *ordered* conditions hold:

(1) $start(e) \geq \lambda(u)$, and
(2) $end(e)$ is minimum.

We can continue on an edge if the start value of the edge is greater than or equal to the label of its starting vertex (which is the end value of the arriving edge). In our railway service representation, this would mean that we can continue on a next train if it departs at or after the time of arrival of our previous train. Of course, this is not true in reality. The only case that we can continue on a train which departs at the same time as the arrival of our previous train, is when these trains are the same one! When the trains are not the same, then the arrival of our previous train and the departure of the next train must be separated by some minimum amount of time: the changing time or connectional margin. In the Dutch railway service network, the connectional margin is not a fixed value. At one station it is 2 minutes, at some other station it is 5 minutes. Even worse, at one single station the connectional margin required for two specific trains may be 2 minutes, while for some other pair of trains it is 5 minutes. At any station the connectional margin may range from 0 minutes (no change) to 5 minutes, depending on the connection.

5.2. Platform dependency

One of the causes of variances in connectional margins is the fact that different trains may arrive on or depart from different platforms. If our connection departs

from the same platform as we arrived on (or in the case of a "cross platform" change), we need little time to change trains. If, in order to go to the departure platform of our next train, we need to walk several stairs, we need more time. It seems promising to represent connectional margins in a similar way. Each vertex (representing a station) is split into several different vertices. For each platform we use one vertex for incoming edges (representing trains arriving at a particular platform), and one vertex for outgoing edges (representing trains departing from a particular platform). If a station has 3 different platforms, it is represented by 6 vertices. Each vertex representing an "arrival platform" is connected by an "inter-platform" edge to all vertices representing "departing platforms". In the case of 3 different platforms we need 9 of these inter-platform edges. The length of the inter-platform edge is made the time that is required to go from one platform to the other. For an example, see fig. 5.1. Obviously, the disadvantage of this approach is that the size of the network increases. If a station has p platforms on average, then the number of vertices becomes $2 * p * |V|$, and the number of edges needed for the inter-platform connections is $p^2 * |V|$.

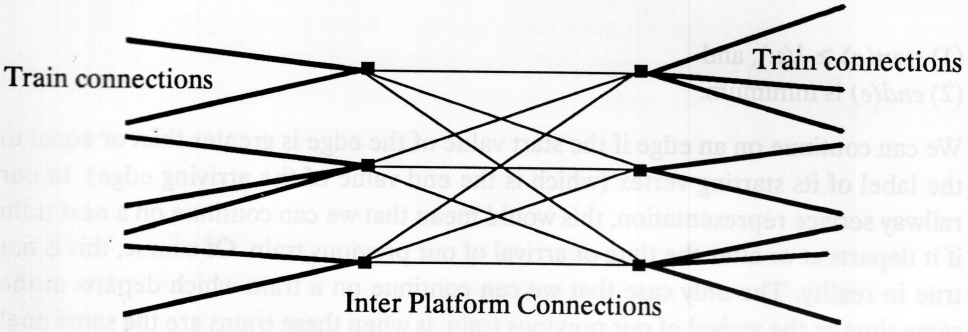


Fig. 5.1.

5.3. Train dependency

Unfortunately, the connectional margins not only depend on the platforms of arrival and successive departure. Another factor which contributes to the time required to change trains is delay sensitivity. We should not forget that the timetables give only the planned times. In reality delays occur. Some connections are guaranteed by the railroad company. If a delay is less than, say, five minutes, then the connecting train is held up. Some connections are not guaranteed however, and although in theory one should be able to make the connection, in practice due to delays this is usually not the case. Even a minimal delay of one single minute may mean a miss. Therefore, these non-guaranteed connections are often given

connectional margins which are greater than the time required for the platform change only. Some margin *specific for the particular connection* is added. Instead of just platform dependent, in reality, the connectional margin may be **train** (edge) dependent. Trying to represent an edge dependent connection cost in the way the platform dependency can be represented as described above, may lead to a representation with one vertex per incoming and one vertex per outgoing edge. This approach is inefficient with regard to network size, as we saw in chapter two.

5.4. A discrete dynamic network

In order to represent an edge dependent connection cost in a discrete network in an adequate and general way, we have developed the concept of a *discrete dynamic network*. In a discrete dynamic network the edge dependent connection cost is represented by a connection function *CON*, which gives the margin required for a specific connection. A discrete dynamic network consists of two parts:

(1) A finite directed graph *(V, E)*, where *V* is the set of vertices and *E* the set of directed edges, each edge of *E* joining two vertices from *V*.
(2) A non-negative, real-valued function, called the connection function *CON*, having three arguments: a vertex and two edges.

With each edge *e* from *E* we associate two values:

(1) a start value *start(e)*;
(2) an end value *end(e)*, satisfying $start(e) < end(e)$.

The length of an edge *e* is defined as:

$$l(e) = end(e) - start(e) .$$

In the representation of a railway service network the vertices represent the stations and the edges represent the trains. The *start* and *end* values of an edge represent the departure and arrival times of the train represented by the edge. The length of the edge represents the travel time of this train. The *CON* function specifies the (minimum) connection times when changing trains.

A *legal path P* from *s* to *t* (in the discrete dynamic network) is an alternating sequence of vertices and edges:

$$s = v_0, e_0, v_1, \ldots, v_j, e_j, v_{j+1}, \ldots, e_{k-1}, v_k = t$$

such that
(1) the start vertex of e_i is v_i and the end vertex of e_i is v_{i+1}, $0 \leq i < k$;
(2) and satisfying the following condition along the path, i.e. for $0 \leq i < k$:

$$start(e_{i+1}) \geq end(e_i) + CON(v_{i+1}, e_i, e_{i+1}) .$$

This condition indicates that if one arrived at vertex v_{i+1} along edge e_i, then it is possible to continue on edge e_{i+1} if the difference between the start value of the outgoing edge, $start(e_{i+1})$, and the end value of the incoming edge, $end(e_i)$, is at least the appropriate connection cost at vertex v_{i+1} (which depends not only on the vertex, but also on the incoming and outgoing edges). This minimum connection cost is $CON(v_{i+1}, e_i, e_{i+1})$. The actual cost of a connection, *COST*, is defined as:

$$COST(v_{i+1}, e_i, e_{i+1}) = start(e_{i+1}) - end(e_i), \text{ with } 0 \leq i < k.$$

Clearly:

$$COST(v_{i+1}, e_i, e_{i+1}) \geq CON(v_{i+1}, e_i, e_{i+1}).$$

The start and end of a (legal) path P are defined as:

$$start(P) = start(e_0) \quad ; \quad end(P) = end(e_{k-1}).$$

While the length of a (legal) path P, $l(P)$, is defined as:

$$l(P) = end(P) - start(P)$$

In the representation of a railway service network, the *CON* function specifies the connectional margin required for a specific train change. The connection *COST* is the time that is actually spent changing to and waiting for the next train. The start and end value of a path are the time of departure and the time of arrival. The length of a path is the duration of the trip, and includes the connection costs. We do not include the time the passenger may have waited at the station at the start of her trip, nor the time she might spent loitering at the arrival station. Time needed for transportation to the station of departure, and from the station of arrival to the final destination of the trip (which is hardly ever the train station), is also not included.

5.5. Space requirements of the CON function

The advantage of the CON function is the space requirement in case of a network with great differences in characteristics of connectional margins. For example, in the Dutch railway services network, many stations have a very simple connectional margin characteristic: 5 minutes in case of a train change, 0 minutes otherwise. For such a station v $CON(v, e_i, e_{i+1})$ becomes a two valued function:

$$\begin{cases} 0 \text{ if } e_i \text{ and } e_{i+1} \text{ represent the same train} \\ 5 \text{ otherwise} \end{cases}$$

Other stations have structural exceptions. For instance: 4 minutes in case of a train change from a 4600 series train to a 14600 series train. For such a station v $CON(v, e_i, e_{i+1})$ becomes a three valued function:

$$\begin{cases} 0 \text{ if } e_i \text{ and } e_{i+1} \text{ represent the same train} \\ 4 \text{ if } e_i \text{ represents a 4600 series train and } e_{i+1} \text{ a series 14600 train} \\ 5 \text{ otherwise} \end{cases}$$

The *CON* can be implemented as an advanced look-up table per vertex. In case of a pure train-to-train dependency, each connection is an entry and its size becomes $|V| * |E| * |E|$, the same as the size of a representation using connecting edges. However, when situations as above occur, its size diminishes significantly. Of course, space is required to store the identification of a train. However, in an application like a railway service information system, this information has to be present anyway.

6. Searching Discrete Dynamic Networks

6.1. Adapting the search algorithm to discrete dynamic networks

In order to construct a search algorithm for a discrete dynamic network, as a basis we use the search algorithm that we have developed for a discrete network (DISNET). In a discrete dynamic network the connection function *CON* is added. The most apparent place to get the *CON* function into the algorithm is to include it in the definition of a relevant edge. Since the *CON* function is dependent on not only the vertex, but also on both the next and the previous edge, we have to know with which edge we labeled a vertex. Therefore we introduce the notion of a *partial path*. In the forward case, a partial path is a legal path from the starting vertex to a non terminating vertex. A *complete path* is a legal path from the starting vertex to the terminating vertex. In the algorithm, the frontier F will consist of partial paths to tentatively labeled vertices, instead of the tentatively labeled vertices themselves. Only those partial paths which gave a vertex its smallest tentative label are considered for further development in the algorithm. We shall now give the DISNET algorithm, in a direct adaptation to discrete dynamic networks.

Consider a discrete dynamic network consisting of the graph $G = (V, E)$ and the connection cost function *CON*. The two special vertices s and t of the network are the *starting* and *terminating* vertices. We want to find a legal path from s to t in our discrete dynamic network, where the end value of the path is minimum and given this end value, the start value of the path is maximum and at least T_{start}.

Pass 1:

(1) $\lambda(s) \leftarrow T_{\text{start}}$ and for all $v \in V, v \neq s,\ \lambda(v) \leftarrow \infty$.
Create a partial path P_0 consisting of s only, $end(P_0) \leftarrow T_{\text{start}}$.
For all $v \in V, \omega(v, u) = \infty$ for each neighbour u of v.
(2) $T \leftarrow V,\ F \leftarrow \{P_0\}$.

(3) Let P_m be a partial path $s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j$ in F for which $end(P_m)$ is minimum; if F is empty then stop, no complete path could be found.
(4) If $u_j = t$, stop, P_m is a complete path with an optimal end value.
(5) If $end(P_m) = \lambda(u_j)$
then for every relevant edge $e_j : u_j \rightarrow u_{j+1}$:
if $u_{j+1} \in T$ and $\lambda(u_{j+1}) > end(e_j)$
then $\lambda(u_{j+1}) \leftarrow end(e_j)$ and create a partial path
$P_n = s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j, e_j, u_{j+1}, F \leftarrow F + \{P_n\}$.
if $end(e_j) < \omega(u_{j+1}, u_j)$
then $\omega(u_{j+1}, u_j) \leftarrow end(e_j)$.
(6) $T \leftarrow T - \{u_j\}$, $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_0, \ldots, u_{j-1}, e_{j-1}, u_j$ and a vertex u_{j+1}, then the relevant edge from u_j to u_{j+1} is the edge e_j: $u_j \rightarrow u_{j+1}$ for which the following two *ordered* conditions hold:

(1) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$, and
(2) $end(e_j)$ is minimum.

In the backward case, a partial path is a legal path from a non-starting vertex to the terminating vertex. The frontier will consist of partial paths from tentatively labeled vertices. Only those partial paths which gave a vertex its highest tentative label are considered for further development in the algorithm. We shall now give the backward pass:

Pass 2:

(1) $\kappa(t) \leftarrow \lambda(t)$ and for all $v \in V, v \neq t$, $\kappa(v) \leftarrow -\infty$.
Create a partial path P_0 consisting of t only, $start(P_0) \leftarrow \lambda(t)$.
(2) $T \leftarrow V$, $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $u_j, \ldots, u_{k-1}, e_{k-1}, t$ in F for which $start(P_m)$ is maximum.
(4) If $u_j = s$, stop, P_m is an optimal complete path.
(5) If $start(P_m) = \kappa(u_j)$
then for every relevant edge $e_{j-1} : u_{j-1} \rightarrow u_j$, with $\omega(u_j, u_{j-1}) \leq \kappa(u_{j-1})$,
if $u_{j-1} \in T$ and $\kappa(u_{j-1}) < start(e_{j-1})$
then $\kappa(u_{j-1}) \leftarrow start(e_{j-1})$ and create a partial path
$P_n = u_{j-1}, e_{j-1}, u_j, \ldots, u_{k-1}, e_{k-1}, t$, $F \leftarrow F + \{P_n\}$.
(6) $T \leftarrow T - \{u_j\}$, $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_j, e_j, u_{j+1}, \ldots, u_k$ and a vertex u_{j-1}, then the relevant edge from u_{j-1} to u_j is the edge e_{j-1}: $u_{j-1} \rightarrow u_j$ for which the following two *ordered* conditions hold:

(1) $end(e_{j-1}) \leq start(e_j) - CON(u_j, e_{j-1}, e_j)$, and
(2) $start(e_{j-1})$ is maximum.

6.2. A counter example: the effect of CON

Unfortunately, it does not suffice to adapt the algorithm in this straightforward way. Because of the differences in values that the *CON* function may return for different connections at the same vertex, in some cases, the algorithm described above will not find an optimal path. For instance, consider the following example (see fig. 6.1). Suppose we want to travel from Utg to Asd, and that we want to depart at or after 7:00. Furthermore, the changing time is 5 minutes for all train changes at Ass (and 0 minutes in the case of no train change).

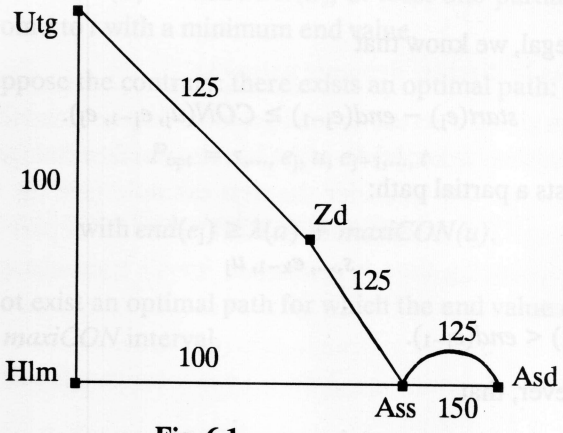


Fig. 6.1.

	100	125	150	Station
Utg	7:00	7:02		Uitgeest
Hlm	7:15	|		Haarlem
Zd	|	7:22		Zaandam
Ass	7:30	7:32	7:40	Amsterdam Sloterdijk
Asd		7:37	7:45	Amsterdam Central Station

In this example, first Utg gets labeled with 7:00. From Utg, Hlm gets labeled with 7:15 and Zd with 7:22. From Hlm, Ass gets labeled 7:30. From Zd, Ass does not get relabeled. Consequently, the partial path from Utg to Ass, arriving at 7:32, will not be developed further (since $\lambda(Ass) = 7{:}30$). The path from Utg to Ass arriving at

7:30 does get developed further and Asd gets labeled with 7:45, since the 7:32 from Ass to Asd may not be used as it is separated from our arrival at Ass by only 2 minutes, while 5 minutes are required. As the earliest possible time of arrival we incorrectly find 7:45. Consequently, the backward pass will incorrectly find as optimal solution a trip from Utg to Ass by train 125 and from Ass to Asd by train 150!

6.3. The invalidity of the Markovian property

The cause of the incorrectness of the algorithm as described above lies in the fact that a discrete dynamic network does not have the Markovian property (see chapter 3). The Markovian property means that if a shortest route from s to t passes through the vertex u, then a shortest path from s to u is part of a shortest path from s to t. However, this is not the case in a discrete dynamic network. Suppose we have an optimal legal path from s to t:

$$P_{opt} = s,\ldots,e_{j-1}, u_j, e_j,\ldots, t.$$

Since the path is legal, we know that

$$start(e_j) - end(e_{j-1}) \geq CON(u_j, e_{j-1}, e_j).$$

Suppose there exists a partial path:

$$s,\ldots, e_{k-1}, u_j$$

such that $end(e_{k-1}) < end(e_{j-1})$.

It may be so, however, that

$$start(e_j) - end(e_{k-1}) < CON(u_j, e_{k-1}, e_j).$$

So,

$$CON(u_j, e_{j-1}, e_j) < CON(u_j, e_{k-1}, e_j)$$

with $end(e_k) < end(e_{j-1})$.

Therefore it is **not** possible to construct the path $s,\ldots, e_{k-1}, u_j, e_j,\ldots, t$, which would have given a new optimal path. Furthermore, the best possible completion of the best partial path: $s,\ldots, e_{k-1}, u_j, e_k,\ldots, t$, may have an end value worse than the end value of the optimal path P_{opt}. In conclusion, a non-optimal partial path to a vertex may

yield an optimal complete path, while an optimal partial path to that vertex cannot be used to construct an equivalent optimal complete path.

6.4. The principle of optimality for discrete dynamic networks

Clearly, an algorithm finding an optimal path in a discrete dynamic network must remember more partial paths per vertex than just the optimal partial path. Fortunately, we can give an upper bound on the end values of the partial paths which need to be remembered (and developed further). We shall refer to this upper bound as the principle of optimality for discrete dynamic networks:

In a discrete dynamic network, let *maxiCON(u)* denote the maximum value that the *CON* function gives for any connection at u (which is non-zero), and let $\lambda(u)$ denote the minimum end value of any partial path from s to u in the network. If there exists a legal path P from s to t in the network, with a minimum end value, and which passes through vertex u, then from the paths from s to u which have an end value which is at least $\lambda(u)$ and less than $\lambda(u) + maxiCON(u)$, at least one partial path is part of a complete path from s to t with a minimum end value.

For the proof, suppose the contrary: there exists an optimal path:

$$P_{opt} = s,..., e_j, u, e_{j+1},..., t$$

$$\text{with } end(e_j) \geq \lambda(u) + maxiCON(u),$$

and there does not exist an optimal path for which the end value of the partial path to u is within the *maxiCON* interval.

We know that

$$end(e_j) \geq \lambda(u) + maxiCON(u).$$

Suppose that the partial path which gave u its label λ is:

$$P_\lambda = s,..., e_k, u.$$

Since by the definition of λ, $end(e_k) = \lambda(u)$, we know that

$$end(e_j) \geq end(e_k) + maxiCON(u).$$

And since $start(e_{j+1})$ must be at least $end(e_j)$, we know that

$$start(e_{j+1}) \geq end(e_k) + maxiCON(u).$$

By the definition of *maxiCON* we know that

$$CON(u, e_k, e_{j+1}) \leq maxiCON(u).$$

So,

$$start(e_{j+1}) \geq end(e_k) + CON(u, e_k, e_{j+1}).$$

But then it is possible to construct an optimal complete path

$$P_{alt} = s,..., e_k, u, e_{j+1},..., t$$

for which the partial path to u arrives within the *maxiCON* interval. Which is a contradiction, since $end(e_k) \leq \lambda(u) + maxiCON(u)$.

6.5. Relevant edges in discrete dynamic networks

Since the edge giving a vertex its smallest label may not lead to an optimal path, while an edge giving it a higher label might, we also need to change the definition of a relevant edge. To see why, consider the next example (see fig. 6.2). Suppose we want to travel from Utg to Asd, and that we want to depart at or after 7:00. Furthermore, the changing time is 5 minutes for all train changes at Ass, and 3 minutes for all train changes at Hlm.

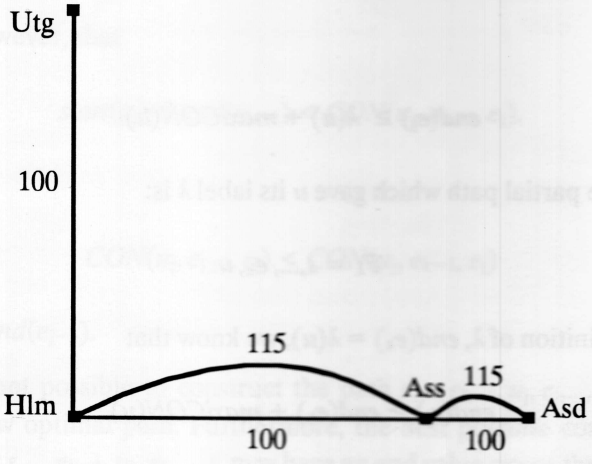


Fig. 6.2.

	100	115	150	Station
Utg	7:00			Uitgeest
Hlm	7:15	7:18		Haarlem
Ass	7:30	7:33	7:40	Amsterdam Sloterdijk
Asd		7:38	7:45	Amsterdam Central Station

In this example, after Hlm is labeled with 7:15, the relevant edge to Ass is the 7:15 arriving at 7:30. Since this train does not travel further to Asd, and since we cannot catch the 7:33 at Ass, we then travel to Asd with train 150, arriving at 7:45. However, we could have changed at Hlm to train 115 which would have taken us to Asd, arriving at 7:38! Although this train was not within the *maxiCON* interval at Hlm, it is within the *maxiCON* interval at Ass. From this example we see that with the current definition of a relevant edge, we could miss edges giving a vertex a label within the $\lambda + maxiCON$ interval. In order to solve this problem, we define a relevant edge as follows: given a partial path $u_0,\ldots, u_{j-1}, e_{j-1}, u_j$ and a vertex u_{j+1}, then the relevant edges from u_j to u_{j+1} are the edges e_j: $u_j \rightarrow u_{j+1}$ for which the following two *ordered* conditions hold:

(1) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$, and
(2) $end(e_j) < end(e_{min}) + maxiCON(u_{j+1})$,
where $end(e_{min})$ is the minimum end value of any edge satisfying (1).

6.6. A search algorithm for discrete dynamic networks: DYNET

Now that we have an upper bound on the partial paths that we need to remember so that we cannot miss the optimal solution, and a matching definition of a relevant edge, we can construct a search algorithm for discrete dynamic networks. We shall refer to this algorithm as the DYNET algorithm.

Since we need to remember and develop further multiple partial paths to a vertex, it may be that one of the partial paths has already been developed further when another partial path reaches that vertex. Therefore, in step (5) of the algorithm, we can no longer test whether a vertex has not been made part of the shortest path tree yet (the test whether the vertex is part of the collection T). A vertex can be part of the shortest path tree multiple times. So, the collection T, which contains all vertices which have not been made part of the shortest path tree, is no longer necessary.

Consider a discrete dynamic network consisting of the graph $G = (V, E)$ and the connection cost function *CON*. The maximum value of *CON* at a vertex u is $maxiCON(u)$, which is non-zero. The two special vertices s and t of the network are

the *starting* and *terminating* vertices. We want to find a legal path from s to t in our discrete dynamic network, where the end value of the path is minimum and given this end value, the start value of the path is maximum and at least T_{start}.

Pass 1:

(1) $\lambda(s) \leftarrow T_{start}$ and for all $v \in V, v \neq s$, $\lambda(v) \leftarrow \infty$.
Create a partial path P_0 consisting of s only, $end(P_0) \leftarrow T_{start}$.
For all $v \in V, \omega(v, u) = \infty$ for each neighbour u of v.
(2) $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j$ in F for which $end(P_m)$ is minimum; if F is empty then stop, no complete path could be found.
(4) If $u_j = t$, stop, P_m is a complete path with an optimal end value.
(5) If $end(P_m) < \lambda(u_j) + maxiCON(u_j)$.
then for every relevant edge $e_j : u_j \rightarrow u_{j+1}$:
if $\lambda(u_{j+1}) > end(e_j)$
then $\lambda(u_{j+1}) \leftarrow end(e_j)$
if $end(e_j) < \lambda(u_{j+1}) + maxiCON(u_{j+1})$
then create a partial path $P_n = s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j, e_j, u_{j+1}$ and
$F \leftarrow F + \{P_n\}$.
if $end(e_j) < \omega(u_{j+1}, u_j)$
then $\omega(u_{j+1}, u_j) \leftarrow end(e_j)$.
(6) $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_0, \ldots, u_{j-1}, e_{j-1}, u_j$ and a vertex u_{j+1}, then the relevant edges from u_j to u_{j+1} are the edges $e_j: u_j \rightarrow u_{j+1}$ for which the following two *ordered* conditions hold:

(1) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$, and
(2) $end(e_j) < end(e_{min}) + maxiCON(u_{j+1})$,
where $end(e_{min})$ is the minimum end value of any edge satisfying (1).

Pass 2:

(1) $\kappa(t) \leftarrow \lambda(t)$ and for all $v \in V, v \neq t$, $\kappa(v) \leftarrow -\infty$.
Create a partial path P_0 consisting of t only, $start(P_0) \leftarrow \lambda(t)$.
(2) $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $u_j, \ldots, u_{k-1}, e_{k-1}, t$ in F for which $start(P_m)$ is maximum.
(4) If $u_j = s$, stop, P_m is an optimal complete path.
(5) If $start(P_m) > \kappa(u_j) - maxiCON(u_j)$
then for every relevant edge $e_{j-1} : u_{j-1} \rightarrow u_j$, with $\omega(u_j, u_{j-1}) \leq start(P_m)$,

if $\kappa(u_{j-1}) < start(e_{j-1})$
then $\kappa(u_{j-1}) \leftarrow start(e_{j-1})$
if $start(e_{j-1}) > \kappa(u_{j-1}) - maxiCON(u_{j-1})$
then create a partial path $P_n = u_{j-1}, e_{j-1}, u_j, \ldots, u_{k-1}, e_{k-1}, t$ and
$F \leftarrow F + \{P_n\}$.

(6) $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_j, e_j, u_{j+1}, \ldots, u_k$ and a vertex u_{j-1}, then the relevant edges from u_{j-1} to u_j are the edges $e_{j-1}: u_{j-1} \rightarrow u_j$ for which the following two *ordered* conditions hold:

(1) $end(e_{j-1}) \leq start(e_j) - CON(u_j, e_{j-1}, e_j)$, and
(2) $start(e_{j-1}) > start(e_{max}) - maxiCON(u_{j-1})$,
where $start(e_{max})$ is the maximum start value of any edge satisfying (1).

6.6.1. The correctness of the forward pass

For the correctness of the algorithm we first prove that when in step (3) of the first pass, a vertex u_j is the end vertex of a partial path for the first time, then we have found a path from s to u_j for which the end value is minimum.

We remark first that since any path but the zero-length path to s is constructed in step (5) of the algorithm, the first time we find a partial path in step (3), arriving at a vertex $u \neq s$, its end value must be equal to $\lambda(u)$.

The first time that we reach step (3), we know that the only path in F is the zero-length path to s with end value T_{start}. Evidently, this end value is minimum.

Suppose at the kth time that we reach step (3), for the first time we find a partial path arriving at u, say the path

$$P_u = s, \ldots, e_j, u, \text{ with an end value } end(e_j) = \lambda(u).$$

Now suppose there exists another partial path from s to u with an end value less than $\lambda(u)$, say the path

$$P_{alt} = s, \ldots, e_{p-1}, v, e_p, u, \text{ with an end value } end(e_p).$$

There are two possibilities: either the vertex v has already been the end vertex of a partial path in step (3) or it has not been. We examine both possibilities in turn:

(1) If v has been the end vertex of a partial path in step (3), then there are again two possibilities: either $\lambda(v) + maxiCON(v) < \lambda(u)$ or $\lambda(v) + maxiCON(v) \geq \lambda(u)$.

If $\lambda(v) + maxiCON(v) < \lambda(u)$, then any path

$$P_v = s,\ldots, e_q, v$$

arriving at v within $\lambda(v) + maxiCON(v)$ must have been selected from F before the path P_u. And by the principle of optimality for discrete dynamic networks, one of those paths could have been used to construct the path

$$P_{v,u} = s,\ldots, e_q, v, e_p, u$$

with an end value of $end(e_p)$.

If $\lambda(v) + maxiCON(v) \geq \lambda(u)$, then since $end(e_p) < \lambda(u)$, it must be true that $end(e_{p-1}) < \lambda(u)$. So, the partial path

$$P_v = s,\ldots, e_{p-1}, v$$

must have been selected from F in step (3) before the path P_u. Consequently, in the following steps this path would have been used to construct the path

$$P_{v,u} = s,\ldots, e_{p-1}, v, e_p, u$$

with an end value of $end(e_p)$.

(2) Now consider the case that v has not yet been the end vertex of a partial path in step (3). Then either the label of v is greater than or equal to the label of u, or it is smaller.

If the label of v is greater than or equal to the label of u, then the path P_{alt} cannot have a smaller end value than $\lambda(u)$.

If the label of v is smaller than the label of u, then the partial path which gave v its label must have been selected from F before P_u, which is a contradiction.

6.6.2. The correctness of the backward pass

For the proof of the backward pass we only need to show that it suffices to construct those paths to neighbours for which $\omega(u_j, u_{j-1}) \leq start(P_m)$ in step (5), where $P_m = u_j, e_j, u_{j+1},\ldots, u_{k-1}, e_{k-1}, t$.

The rest of the proof is similar to the proof of the forward pass.

From the correctness of the forward pass we know that there does not exist a partial path

$$P_s = s,..., u_{k-1}, e_{k-1}, u_{j-1}, e_k, u_j$$

for which the end value is smaller than $\omega(u_j, u_{j-1})$, and the start value is at least T_{start}. So, if

$$\omega(u_j, u_{j-1}) > start(P_m) = start(e_j),$$

then we know that for every path P_s with a start value of at least T_{start}

$$end(e_k) > start(e_j).$$

But then surely for every path P_s with a start value of at least T_{start}

$$end(e_k) + CON(u_j, e_k, e_j) > start(e_j).$$

So, there does not exist a path P_s with a start value of at least T_{start}, which can be used to construct a legal path with P_m. Therefore we could never develop an optimal path out of P_m by using neighbour u_{j-1}.

Since we have proven that the first pass gives the smallest possible end value, and that the backward pass gives the highest matching start value, the correctness of the algorithm follows.

6.7. Dynamic generation of vertices

What the algorithm does conceptually speaking, is dynamically generating a vertex for each arriving edge. Each partial path arriving at a vertex u can be seen as a vertex for the arriving edge. The edge connecting the vertex to departing edges is (implicitly) represented by the connection cost function *CON*. The departing edge is implicitly represented by the vertex representing the arrival of this edge. In the algorithm, only those vertices are generated, which represent arriving trains which may be of interest for an optimal solution. This approach is obviously more efficient with regard to memory usage, than the vertex representation discussed in the previous chapter.

7. General Dynamic Networks

Visiting costs as described in the previous chapter are not restricted to discrete networks. In this chapter we shall look at an example where visiting costs occur in a ordinary weighted graph. To represent visiting costs in an ordinary graph adequately, we propose a dynamic network.

7.1. An example application

If we want to represent a network of roads we can use a weighted graph. The vertices of the graph represent cities and junctions. The edges of the graph represent the roads. The length of an edge representing a road is the driving time under normal driving conditions (not including stops and observing speed limits). A road may be for instance a state highway, a provincial highway or a secondary road. Driving times may vary depending on properties such as the class of the road, the number of lanes and the road surface. Furthermore, extra time may be required when one changes highways or roads. Changing roads may involve using exit and entry lanes, roundabouts or connecting roads within a city. Consider the following example (see fig. 7.1), which is an (imaginary) junction:

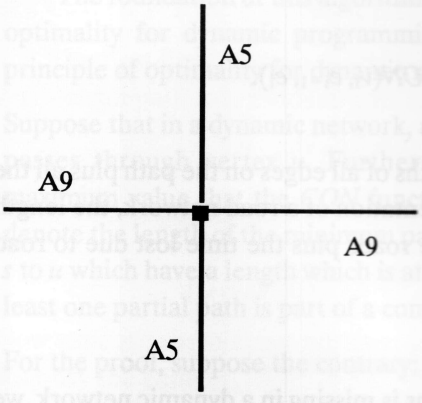


Fig. 7.1.

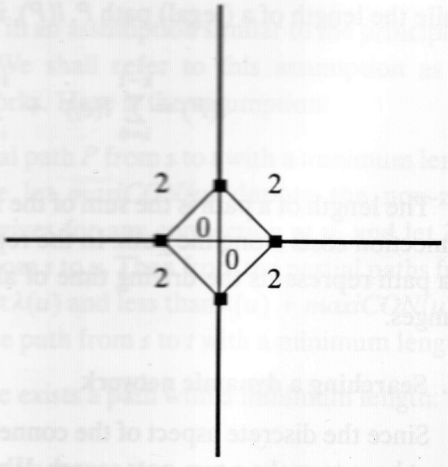


Fig. 7.2.

If it takes 2 minutes to change from the A9 to the A5 at the junction (due to connection characteristics such as traffic lights or due to the decelerating and successive accelerating), then we cannot add those two minutes to the length of either the A5 or A9 section, since the cost is not applicable if one travels through on either the A5 or A9 at the junction. If multiple roads with different connection characteristics exist between two junctions, then the changing cost may be different for each road, yielding an edge dependent visiting cost. Rather than using an edge between the appropriate sections as in fig. 7.2, which requires 4 vertices and 6 edges in the case of a crossing of two roads, we propose to use a connection function.

7.2. Dynamic networks

A dynamic network consists of two parts:

(1) A finite weighted graph $G = (V, E)$, where V is the set of vertices and E the set of edges, each edge of E joining two vertices from V is assigned a non-negative length.
(2) A non-negative, real-valued function, called the connection function CON, having three arguments: a vertex and two edges.

A *legal path* P from s to t (in the graph) is an alternating sequence of vertices and edges:

$$s = v_0, e_0, v_1, \ldots, v_j, e_j, v_{j+1}, \ldots, e_{k-1}, v_k = t,$$

such that the start vertex of e_i is v_i and the end vertex of e_i is v_{i+1}, $0 \leq i < k$.

While the length of a (legal) path P, $l(P)$, is defined as:

$$l(P) = \sum_{i=0}^{k-1} l(e_i) + \sum_{i=1}^{k-1} CON(v_i, e_{i-1}, e_i).$$

The length of a path is the sum of the lengths of all edges on the path plus all the connection costs along the path. In the representation of a road network, the length of a path represents the driving time of all the roads plus the time lost due to road changes.

7.3. Searching a dynamic network

Since the discrete aspect of the connections is missing in a dynamic network, we do not have to make a two-pass search. We can adapt the algorithm we developed for

a discrete dynamic network, leaving out those aspects which dealt with the discreteness of the connections and the two passes. Here is a search algorithm for a dynamic network:

Consider a dynamic network consisting of the graph $G = (V, E)$ and the connection cost function *CON*. The maximum value of *CON* at a vertex u is *maxiCON(u)*, which is non-zero. The two special vertices s and t of the network are the *starting* and *terminating* vertices. We want to find a legal path from s to t in our dynamic network, where the length of the path is minimum.

(1) $\lambda(s) \leftarrow 0$ and for all $v \in V, v \neq s, \lambda(v) \leftarrow \infty$.
Create a partial path P_0 consisting of s only, $end(P_0) \leftarrow 0$.
(2) $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j$ in F for which $l(P)$ is minimum; if F is empty then stop, no path could be found.
(4) If $u_j = t$, stop, P_m is a path of minimum length.
(5) If $end(P_m) < \lambda(u_j) + maxiCON(u_j)$.
then for every edge $e_j : u_j \rightarrow u_{j+1}$:
if $\lambda(u_{j+1}) > \lambda(u_j) + l(e_j) + CON(u_j, e_{j-1}, e_j)$
then $\lambda(u_{j+1}) \leftarrow \lambda(u_j) + l(e_j) + CON(u_j, e_{j-1}, e_j)$.
if $\lambda(u_j) + l(e_j) + CON(u_j, e_{j-1}, e_j) < \lambda(u_{j+1}) + maxiCON(u_{j+1})$
then create a partial path $P_n = s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j, e_j, u_{j+1}$ and
$F \leftarrow F + \{P_n\}$.
(6) $F \leftarrow F - \{P_m\}$ and go to step (3).

7.3.1. The principle of optimality for dynamic networks

The foundation of this algorithm lies in an assumption similar to the principle of optimality for dynamic programming. We shall refer to this assumption as the principle of optimality for dynamic networks. Here is the assumption:

Suppose that in a dynamic network, a legal path P from s to t with a minimum length passes through vertex u. Furthermore let *maxiCON(u)* denote the non-zero maximum value that the *CON* function gives for any connection at u, and let $\lambda(u)$ denote the length of the minimum path from s to u. Then from the partial paths from s to u which have a length which is at least $\lambda(u)$ and less than $\lambda(u) + maxiCON(u)$, at least one partial path is part of a complete path from s to t with a minimum length.

For the proof, suppose the contrary: there exists a path with a minimum length:

$$P_{opt} = s, \ldots, e_j, u, e_{j+1}, \ldots, t.$$

We divide the path into two partial paths:

$$P_{s,u} = s,..., e_j, u \quad ; \quad P_{u,t} = u, e_{j+1},..., t.$$

Clearly,

$$l(P_{opt}) = l(P_{s,u}) + CON(u, e_j, e_{j+1}) + l(P_{u,t}),$$

Suppose that for the length of the partial path $l(P_{s,u}) \geq \lambda(u) + maxiCON(u)$, and there does not exist a path with a minimum length for which the length of the partial path to u is within the *maxiCON* interval.

We know that

$$l(P_{s,u}) \geq \lambda(u) + maxiCON(u).$$

Suppose that the partial path which gave u its label λ is:

$$P_\lambda = s,..., e_k, u.$$

Since by definition $l(P_\lambda) = \lambda(u)$, we know that

$$l(P_\lambda) + maxiCON(u) \leq l(P_{s,u}).$$

By the definition of *maxiCON* we know that

$$CON(u, e_k, e_{j+1}) \leq maxiCON(u).$$

So,

$$l(P_\lambda) + CON(u, e_k, e_{j+1}) \leq l(P_{s,u}),$$

then surely

$$l(P_\lambda) + CON(u, e_k, e_{j+1}) \leq l(P_{s,u}) + CON(u, e_j, e_{j+1}).$$

But then it is possible to construct the following complete path

$$P_{alt} = s,..., e_k, u, e_{j+1},..., t.$$

For which,

$$l(P_{alt}) = l(P_{\lambda}) + CON(u, e_k, e_{j+1}) + l(P_{u,t}),$$

Since,

$$l(P_{\lambda}) + CON(u, e_k, e_{j+1}) + l(P_{u,t}) \leq l(P_{s,u}) + CON(u, e_j, e_{j+1}) + l(P_{u,t}),$$

We get

$$l(P_{alt}) \leq l(P_{opt}).$$

Since $l(P_{opt})$ is minimum, it must be that

$$l(P_{alt}) = l(P_{opt}).$$

So, the length of this new path P_{alt} must be minimum while the length of its partial path to u is less then $\lambda(u) + maxiCON(u)$. Which is a contradiction.

7.3.2. The correctness of the algorithm

For the correctness of the algorithm we proof that when in step (3) of the algorithm, a vertex u_j is the end vertex of a partial path for the first time, then we have found a path of minimum length from s to u_j.

We remark first that, since any path but the zero-length path to s is constructed in step (5) of the algorithm, the first time we find a partial path in step (3), arriving at a vertex $u \neq s$, its length must be equal to $\lambda(u)$.

The first time that we reach step (3), we know that the only path in F is the zero-length path to s. Evidently, its length is minimum.

Suppose at the kth time that we reach step (3), for the first time we find a partial path arriving at u, say the path

$$P_u = s,..., e_j, u, \text{ with a length of } \lambda(u).$$

Now suppose there exists another partial path from s to u with a length less than $\lambda(u)$, say the path

$$P_{alt} = s,..., e_{p-1}, v, e_p, u, \text{ with a length } l(P_{alt}).$$

There are two possibilities: either the vertex v has already been the end vertex of a partial path in step (3) or it has not been. We examine both possibilities in turn:

(1) If v has been the end vertex of a partial path in step (3), then there are again two possibilities: either $\lambda(v) + maxiCON(v) < \lambda(u)$ or $\lambda(v) + maxiCON(v) \geq \lambda(u)$.

If $\lambda(v) + maxiCON(v) < \lambda(u)$, then any path

$$P_v = s,..., e_q, v,$$

with a length less than $\lambda(v) + maxiCON(v)$, must have been selected from F before the path P_u. And by the principle of optimality for dynamic networks, one of those paths could have been used to construct the path

$$P_{v,u} = s,..., e_q, v, e_p, u.$$

So, we would have found a path with a length of $L(P_{alt})$ then.

If $\lambda(v) + maxiCON(v) \geq \lambda(u)$, then since it must be true that $l(P_v) < \lambda(u)$, the length of the partial path

$$P_{s,v} = s,..., e_{p-1}, v$$

must satisfy

$$l(P_{s,v}) < \lambda(u).$$

So, $P_{s,v}$ must have been selected from F in step (3) before the path P_u. Consequently, in the following steps this path would have been used to construct the path P_{alt}.

(2) Now consider the case that v has not yet been the end vertex of a partial path in step (3). Then either the label of v is greater than or equal to the label of u, or it is smaller.

If the label of v is greater than or equal to the label of u, then the path P_{alt} cannot possibly have smaller length than $\lambda(u)$.

If the label of v is smaller than the label of u, then the partial path which gave v its label must have been selected from F before P_u, which is a contradiction.

7.3.3. Dynamic generation of vertices

As with the algorithm for searching a discrete dynamic network, what the algorithm does conceptually speaking, is dynamically generating a vertex for each arriving edge. The edge connecting the vertex to departing edges is (implicitly) represented by the connection cost function *CON*. In the algorithm, only those vertices are generated, which represent arriving edges which may be of interest for an optimal solution.

8. Space Reduction Method

When searching, it may often prove to be more efficient to first reduce the size of the search space and then to search that reduced space, instead of just searching the entire, initial search space. The gain will be increased if the search space will be searched several times, perhaps to find solutions with different characteristics. There is an obvious danger: when cutting the search space, we must be careful not to eliminate the optimal solutions to the original problem, or more generally, the interesting solutions to this problem. We describe the Space Reduction Method SRM, which reduces a search space without losing optimal solutions. SRM is particularly applicable to searching graphs in which pairs of vertices are connected by several parallel edges. SRM is independent of the search method used. We shall show how it can be used on a railway service network, an example of a discrete (dynamic) network. This chapter is largely similar to [Si, 1991].

8.1. Domains of application

SRM can be applied to any domain which can be represented by a graph in which often several parallel edges occur. A transportation service network, represented by a discrete network (or possibly a discrete dynamic network), is an example of such a domain. The stops are the vertices of the network, while the edges are particular transportation services linking two vertices. For example, the vertices may be the stations Utrecht CS and Woerden, the edges the trains connecting these stations as in fig 8.1 (a small excerpt of the 1989-1990 Dutch time-tables). In fig. 8.1, foot-note (1) means that a train does not run on Saturdays, Sundays and public holidays, December 27th, 28th, and 29th, April 13th, and May 25th. The "two hammers" sign means that a train runs on weekdays and Saturdays only. An encircled A means that a train runs on weekdays only, a † means that a train runs on Sundays and public holidays only. A bus service network and an air service network are similar to a railway service network.

13 b

km	treinnummer		9806	9910	9810 5323	19912 *5325*	9912	9812 5327	13512	19012 5527	9914	9814 5329
0	**Utrecht CS**		Ⓐ 5 17	Ⓐ 5 37	Ⓐ 5 57		⚒ 6 15	⚒ 6 20		Ⓐ 6 48	Ⓐ 6 50	† 7 01
7	Vleuten		Ⓐ |	5 43	Ⓐ 6 03		6 21	⚒ |		Ⓐ |	6 56	† 7 07
16	Woerden	A	Ⓐ 5 27	5 49	Ⓐ 6 09		6 27	⚒ 6 30		Ⓐ 6 58	7 02	† 7 12
16	Woerden			5 49			6 27				7 02	
27	Bodegraven			5 57			6 34				7 12	
35	**Alphen a/d Rijn**	A		6 03			6 40				7 18	
35	**Alphen a/d Rijn**			6 04		1 6 26	6 47		1 7 05		7 21	
50	Leiden Lammenschans			6 12		1 6 35	6 55		1 7 13		7 29	
50	**Leiden**	A		Ⓐ 6 17		1 6 40	⚒ 7 00		1 7 18		Ⓐ 7 34	
	Leiden				Ⓐ 6 22	*6 48*		7 16		⚒ 7 26		7 49
	Den Haag CS	A			Ⓐ 6 38	*7 04*		7 33		⚒ 7 42		8 05

13 b > vervolg >

km	treinnummer		9924	9826 5537	9926	9828 5539	9928	9830 5541	9930	9832 5543	9932	9834 5545
0	**Utrecht CS**		9 10	9 33	9 40	10 03	10 09	10 33	10 40	11 03	11 10	11 33
7	Vleuten		9 16	|	9 46	|	10 16	|	10 46	|	11 16	|
16	Woerden	A	9 22	9 43	9 52	10 13	10 22	10 43	10 52	11 13	11 22	11 43
16	Woerden		9 22		9 52		10 22		10 52		11 22	
27	Bodegraven		9 29		9 59		10 29		10 59		11 29	
35	**Alphen a/d Rijn**	A	9 35		10 05		10 35		11 05		11 35	
35	**Alphen a/d Rijn**		9 36		10 06		10 36		11 06		11 36	
50	Leiden Lammenschans		9 44		10 14		10 44		11 14		11 44	
50	**Leiden**	A	9 49		10 19		10 49		11 19		11 49	
	Leiden			10 02		10 32		11 02		11 32		12 02
	Den Haag CS	A		10 18		10 48		11 18		11 48		12 18

13 b > vervolg >

km	treinnummer		9848 5559	9948	9850 5561	9950 *5563*	19952	9852 5365	9952	9854 5565	19954 *5367*	9954 *5567*
0	**Utrecht CS**		15 03	15 10	15 33	15 40		16 03	16 10	16 33		16 40
7	Vleuten		|	15 16	|	15 46		|	16 16	|		16 46
16	Woerden	A	15 13	15 22	15 43	15 52		16 13	16 22	16 43		16 52
16	Woerden			15 22		15 52			16 22			16 52
27	Bodegraven			15 29		15 59			16 29			16 59
35	**Alphen a/d Rijn**	A		15 35		16 05			16 35			17 05
35	**Alphen a/d Rijn**			15 36		16 06	1 16 22		16 36		1 16 52	17 06
50	Leiden Lammenschans			15 44		16 14	1 |		16 44		1 |	17 14
50	**Leiden**	A		15 49		16 19	1 16 33		16 49		1 17 03	17 19
	Leiden		15 32		16 02	*16 32*		16 48		17 02	*17 18*	*17 32*
	Den Haag CS	A	15 48		16 18	*16 48*		17 05		17 18	*17 35*	*17 48*

13 b > vervolg >

km	treinnummer		9966	9868 5579	9968	9870 5581	9970	9872 5583	9972	9874 5585	9974	9876 5587
0	**Utrecht CS**		19 40	20 03	20 10	20 33	20 40	21 03	21 10	21 33	21 40	22 03
7	Vleuten		19 46	|	20 16	|	20 46	|	21 16	|	21 46	|
16	Woerden	A	19 52	20 13	20 22	20 43	20 52	21 13	21 22	21 43	21 52	22 13
16	Woerden		19 52		20 22		20 52		21 22		21 52	
27	Bodegraven		19 59		20 29		20 59		21 29		21 59	
35	**Alphen a/d Rijn**	A	20 05		20 35		21 05		21 35		22 05	
35	**Alphen a/d Rijn**		20 06		20 36		21 06		21 36		22 06	
50	Leiden Lammenschans		20 14		20 44		21 14		21 44		22 14	
50	**Leiden**	A	20 19		20 49		21 19		21 49		22 19	
	Leiden			20 32		21 02		21 32		22 02		22 32
	Den Haag CS	A		20 49		21 18		21 48		22 18		22 48

Fig. 8.1 An excerpt from the 1989 - 1990 Dutch time-tables.

	9814	9916 *5529*	19918	9816	9818	19018	9918	9820 5531	19920 *5333*	9920	9822 5533	9922	9824 5535
UC	⚒ 7 02	⚒ 7 06		⚒ 7 11	⚒ 7 29	⚒ 7 32	7 36	8 03		8 10	8 33	8 40	9 03
Vl	⚒ |	7 12		⚒ |	⚒ 7 35	⚒ |	7 42	|		8 16	|	8 46	|
Wo	⚒ 7 12	7 18		⚒ 7 21	⚒ 7 41	⚒ 7 44	7 48	8 13		8 22	8 43	8 52	9 13
Wo		7 19					7 49			8 22		8 52	
Bo		7 28					7 59			8 29		8 59	
AR		7 34					8 05			8 35		9 05	
AR		7 37	1 7 52				8 06		1 8 22	8 36		9 06	
LL		7 45	8 00				8 14		1 8 30	8 44		9 14	
Le		⚒ 7 50	8 06				8 19		1 8 35	8 49		9 19	
Le		*8 01*	8 11					8 32	*8 48*		9 02		9 32
HC		*8 18*	1 8 26					8 48	*9 05*		9 18		9 48

	9934	9836 5547	9936	9838 5549	9938	9840 5551	9940	9842 5553	9942	9844 5555	9944	9846 5557	9946
UC	11 40	12 03	12 10	12 33	12 40	13 03	13 10	13 33	13 40	14 03	14 10	14 33	14 40
Vl	11 46	|	12 16	|	12 46	|	13 16	|	13 46	|	14 16	|	14 46
Wo	11 52	12 13	12 22	12 43	12 52	13 13	13 22	13 43	13 52	14 13	14 22	14 43	14 52
Wo	11 52		12 22		12 52		13 22		13 52		14 22		14 52
Bo	11 59		12 29		12 59		13 29		13 59		14 29		14 59
AR	12 05		12 35		13 05		13 35		14 05		14 35		15 05
AR	12 06		12 36		13 06		13 36		14 06		14 36		15 06
LL	12 14		12 44		13 14		13 44		14 14		14 44		15 14
Le	12 19		12 49		13 19		13 49		14 19		14 49		15 19
Le		12 32		13 02		13 32		14 02		14 32		15 02	
HC		12 48		13 18		13 48		14 18		14 48		15 18	

	19956	9856 5369	9956	9858 5569	19958 *5371*	9958	9860 5571	9960	9862 5573	9962	9864 5575	9964	9866 5577
UC		17 03	17 10	17 33		17 40	18 03	18 10	18 33	18 40	19 03	19 10	19 33
Vl		|	17 16	|		17 46	|	18 16	|	18 46	|	19 16	|
Wo		17 13	17 22	17 43		17 52	18 13	18 22	18 43	18 52	19 13	19 22	19 43
Wo			17 22			17 52		18 22		18 52		19 22	
Bo			17 29			17 59		18 29		18 59		19 29	
AR			17 35			18 05		18 35		19 05		19 35	
AR	1 17 22		17 36		1 17 52	18 06		18 36		19 06		19 36	
LL	1 |		17 44		1 |	18 14		18 44		19 14		19 44	
Le	1 17 33		17 49		1 18 03	18 19		18 49		19 19		19 49	
Le		17 48		18 02	*18 18*		18 32		19 02		19 32		20 02
HC		18 05		18 18	*18 35*		18 48		19 18		19 48		20 18

	9976	9878 5589	9978	9880 5591	9980	9882 5593	9984	4089 5097
UC	22 10	22 33	22 40	23 03	23 10	23 33	0 07	0 22
Vl	22 16	|	22 46	|	23 16	23 39	0 13	0 28
Wo	22 22	22 43	22 52	23 13	23 22	23 45	0 19	0 35
Wo	22 22		22 52		23 22		0 19	
Bo	22 29		22 59		23 29		0 26	
AR	22 35		23 05		23 35		0 32	
AR	22 36		23 06		23 37		0 38	
LL	22 44		23 14		23 45		0 46	
Le	22 49		23 19		23 50		0 51	
Le		23 02		23 32		0 02		1 02
HC		23 18		23 48		0 18		1 22

< einde <

8.2. The algorithm for SRM

Suppose we want to search a certain discrete (dynamic) network for an optimal path from the source vertex s to the terminating vertex t. The network may be large because there are many vertices, or because many parallel edges occur. In these cases we may want to use SRM to speed up search. In SRM we first search an abstracted version of the network in order to determine which vertices lie on paths which can result in optimal solutions in the "real" network. The steps in SRM are the following:

(1) { Section 8.3.1: The Idealized Skeleton Graph ISG } The discrete network is transformed into its Idealized Skeleton Graph, ISG. Effectively, the Idealized Skeleton Graph is a weighted directed graph in which only a shortest of the (parallel) edges joining two vertices in the discrete network (together with its length) is present. The information about the *start* and *end* values of edges is not kept.

(2) { Section 8.3.2: The Idealized Solution IS } We search for an optimal solution IS from s to t in ISG. Let the length of this optimal path be $l(IS)$.

(3) { Section 8.3.3: Loosening the solution in ISG } In ISG, we find all vertices which are on paths joining s and t, of length less than $(1 + p) * l(IS)$, where $p \geq 0$. The set of these vertices is V': the reduced set of vertices. Note that s and t are in V'.

(4) { Section 8.3.4: The Reduced Graph G' } The set of edges E is reduced to E', which contains those edges from E which join vertices from V'. We thereby construct the reduced graph $G' = (V', E')$.

(5) { Section 8.3.5: The search in the Reduced Graph G': first pass } We search for the optimal solution from s to t in G'. Let its length be $(1 + q) * l(IS)$. Note that $q \geq 0$. If no solution is found, SRM has failed to define a feasible reduced graph, and we may have to resort to searching the entire graph G.

(6) { Section 8.3.6: Verification of optimality } If $q \leq p$, we have found an optimal solution.

(7) { Section 8.3.7: Repair: second pass } Otherwise, if $q > p$, we may, or may not. To verify and eventually find an optimal solution, we replace p by q and go back to step (3). We then obtain a (possibly new) set of vertices $V'' \supseteq V'$, and a new set of edges $E'' \supseteq E'$. When searching the corresponding graph $G'' = (V'', E'')$, the solution found in step (5) will definitely be optimal.

8.3. The different steps in SRM

We now describe and illustrate each step of the space reduction method.

8.3.1. The Idealized Skeleton Graph ISG

In this step, throughout G we replace all the edges from E joining two vertices (the parallel edges), by one single edge which has the same length as the shortest of the parallel edges (if there exists one edge only, this edge is replaced by an edge with the same length). In this way we create a new set of edges E_{ISG}, and define the Idealized Skeleton Graph ISG $= (V, E_{ISG})$. Here is the algorithm to create E_{ISG}:

Consider a discrete network $G = (V, E)$, where V is the set of vertices and E the set of directed edges. Each edge e from E has a start value *start(e)* and an end value *end(e)*, with *start*(e) < *end*(e). We shall construct the Idealized Skeleton Graph ISG $= (V, E_{ISG})$.

(1) $E_{ISG} \leftarrow \varnothing$
(2) For each edge e: $u \rightarrow v$ in E:
If there does not exist an edge e' in E_{ISG} such that e': $u \rightarrow v$,
or if it does exist and *end*(e) − *start*(e) < $l(e')$.
Then create an edge e_{ISG}: $u \rightarrow v$, $l(e_{ISG}) \leftarrow$ *end*(e) − *start*(e),
$E_{ISG} \leftarrow E_{ISG} + \{e_{ISG}\}$.

In our example of fig. 8.1, the 37 edges (directly) connecting Utrecht CS and Woerden in the discrete network representing this time-table, are replaced by one edge with a length of 10 in ISG. It is not important to know which edge in the discrete network induced the edge in ISG, nor the exact *start* and *end* values of this edge, nor the fact that there may also exist a longer edge (in our example an edge with length 12, representing train 19018 departing at 7:32). We only need to know that there exists at least one edge connecting Utrecht CS and Woerden, with a length of 10, and no shorter ones.

Note that $|E_{ISG}| \leq |E|$; the vertices in G and ISG are are the same. The Idealized Skeleton Graph is built only once for a fixed network. Also note that ISG is a weighted, directed graph whose edges have a non-negative length.

8.3.2. The Idealized Solution

In ISG we search for an optimal path from s to t. We call a solution to this problem the Idealized Solution. Since ISG is a directed, weighted graph, we can use any suitable graph search algorithm, for instance Dijkstra's algorithm, see [Di, 1959]. Let the Idealized Solution be the path IS, with length $l(IS)$. Note that the solution represented by the path in ISG may not be realizable in reality (i.e. in G). In our example of fig. 8.1, if the Idealized Solution includes the edge from Utrecht CS to Woerden, the underlying assumption is that we travel from Utrecht CS to Woerden

by a direct train, taking 10 minutes. However, if we would arrive at Utrecht CS at 10:05, we just missed the 10:03 direct train and we would take the 10:09 stopping train, taking 13 minutes for the trip. Note that this last train is represented in ISG by two edges: one from Utrecht to Vleuten with length 6 (induced by, for example train 9910), and one from Vleuten to Woerden with length 5 (induced by train 9814). The Idealized Solution does not take into account the exact times of departure or arrival (*start* and *end* values in a discrete network), time margins necessary to change trains (the CONnection function in a discrete dynamic network), nor other restrictions (such as foot-notes of trains); it is the absolutely best solution one could hope for, and is often not realizable.

8.3.3. Loosening the solution in ISG

Since the Idealized Solution is usually not realizable in G, we loosen it: we consider all solutions in ISG with a length larger than $l(IS)$, but smaller than L_{limit}, where:

$$L_{\text{limit}} = (1 + p) * l(IS), \text{ where } p \geq 0.$$

For example, we might take $p = 0.5$. The choice of the size of p will be discussed below. When we compute in ISG all solutions of length at most L_{limit}, we gather the vertices on the paths representing these solutions in the set V': the reduced set of vertices.

For the computation of these solutions, we need to know all paths from s to t, of length at most L_{limit}. Explicitly constructing all these paths may be very time consuming, since these paths need not be disjunct.

There is a way to find all the vertices we need without explicitly constructing all possible paths. We first conduct a search from the starting vertex s to determine all vertices u for which there exists path from s to u with length less than or equal to L_{limit}. For instance, we can use Dijkstra's algorithm, stopping when the label of the vertex that is made permanent in step (3) (see chapter 3) is greater than L_{limit}. Then we conduct a backward search from the terminating vertex t to determine all vertices v for which there exists a path from v to t with length less than or equal to L_{limit}. In this second search we only need to consider those vertices which were made permanent in the search from s. For each vertex u that was not made permanent in the first search, there does not exist a path from s to u with a length less than or equal to L_{limit}. So there cannot exist such a path from s to t via u. If, in the algorithm, we initialize $\delta_s(u)$ (the length of the shortest path from the source vertex s to the vertex u) to ∞ for every vertex u, then every vertex that was made permanent in the first pass, has a δ_s value of at most L_{limit}. So, of each vertex v that was made permanent in

the second search (and thus in *both* searches), we know the length of a shortest path from the source vertex s to v, $\delta_s(v)$, and the length of a shortest path from v to the terminating vertex t, $\delta_t(v)$. Then we select those vertices v for which $\delta_s(v) + \delta_t(v) \leq L_{\text{limit}}$. Clearly, only those vertices lie on a path from s to t with a length less than or equal to L_{limit}. Note, however, that these paths need not be simple paths! Therefore, the resulting search space may contain dead-end branches. Here is the complete algorithm:

Pass 1:

(1) $\delta_s(s) \leftarrow 0$ and for all $v \in V, v \neq s$, $\delta_s(v) \leftarrow \infty$.
(2) $T \leftarrow V$, $F \leftarrow \{s\}$.
(3) If F is empty then stop. Otherwise, let u be a vertex in F for which $\delta_s(u)$ is minimum.
(4) If $\delta_s(u) > L_{\text{limit}}$, stop.
(5) For every edge $e: u \rightarrow v$,
    if $v \in T$ and $\delta_s(v) > \delta_s(u) + l(e)$
    then $\delta_s(v) \leftarrow \delta_s(u) + l(e)$ and $F \leftarrow F + \{v\}$.
(6) $T \leftarrow T - \{u\}$, $F \leftarrow F - \{u\}$ and go to step (3).

Pass 2:

(1) $\delta_t(t) \leftarrow 0$ and for all $v \in V, v \neq t$, $\delta_t(v) \leftarrow \infty$. $V' \leftarrow \varnothing$.
(2) $T \leftarrow V$, $F \leftarrow \{t\}$.
(3) If F is empty then stop. Otherwise, let u be a vertex in F for which $\delta_t(u)$ is minimum.
(4) If $\delta_t(u) > L_{\text{limit}}$, stop.
(5) If $\delta_s(u) + \delta_t(u) \leq L_{\text{limit}}$ then $V' \leftarrow V' + \{u\}$.
(6) For every edge $e: v \rightarrow u$,
    if $v \in T$ and $\delta_s(v) \leq L_{\text{limit}}$ and $\delta_t(v) > \delta_t(u) + l(e)$
    then $\delta_t(v) \leftarrow \delta_t(u) + l(e)$ and $F \leftarrow F + \{v\}$.
(7) $T \leftarrow T - \{u\}$, $F \leftarrow F - \{u\}$ and go to step (3).

8.3.4. The Reduced Graph G'

We now construct the Reduced Graph $G' = (V', E')$. The collection of vertices of G' is the set V' as defined in the previous section. We construct the reduced set of

edges E' by taking the edges from E, which join vertices from V'. An edge from E is placed in E' iff both its start and end vertex are in V':

(1) $E' \leftarrow \varnothing$.
(2) For every edge $e: u \rightarrow v$ from E:
If $u \in V'$ and $v \in V'$
Then $E' \leftarrow E' + \{e\}$.

In our example of fig. 8.1, if we were looking for optimal solutions from Utrecht CS to Bodegraven, V' would surely include Woerden, but not Maastricht (Maastricht is typically 178 kilometers and 2 hours from Utrecht CS and 205 kilometers and 2:30 hrs from Bodegraven). In E' we would have all trains connecting any two stations in V'. Obviously, the network G' is a restriction of G "around" the Idealized Solution. So, the graph G' is of the same type as G, in our case a discrete (dynamic) network. It is likely that the actual best solution (in G) will be in G'. In addition, G' will often be very much smaller than G.

8.3.5. The search in the Reduced Graph G': first pass

In G' we search for an optimal path from s to t. Let this solution be *Tent.Sol.* (Tentative Solution). This search can be carried out by using any particular search technique suitable for the type of graph (of G' and G), as long as it guarantees an optimal solution. In our case we could use the algorithm from chapter 4 (for a discrete network) or 6 (for a discrete dynamic network). If no solution is found, the Idealized Solution was based on information which was (far) too unrealistic, and cannot be used to define a feasible reduced graph, and we may have to resort to searching the entire graph G. In this case SRM is not successful in reducing the size of the search space.

8.3.6. Verification of Optimality

Let the length of *Tent.Sol.* be $(1 + q) * l(IS)$, where obviously $q \geq 0$. Remember that *Tent.Sol.* is a "real" solution, i.e. it describes a path in G', and thus in G, from s to t. However, although the path is optimal in G', we do not yet know whether it is also optimal in G.

If $q \leq p$, then *Tent.Sol.* is also an optimal solution in G, i.e. by searching only in the reduced space G', we still found an optimal solution for the search in the entire space G, as we now shall prove by contradiction. Let us assume that there is in G a better solution:

$$Sol2 : s = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \ldots u_{n-1} \rightarrow u_n = t$$

which is not entirely in G', i.e.

$$\exists\, i,\ 0 < i < n, \text{ such that } u_i \notin G'.$$

Let the length of *Sol2* (which is a better solution than *Tent.Sol.*) be:

$$l(Sol2) = (1 + r) * l(IS), \text{ where } 0 \leq r < q \leq p.$$

Now let us consider the corresponding path of *Sol2* in ISG:

$$Sol2_{ISG} : s = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \ldots u_{n-1} \rightarrow u_n = t.$$

Since in ISG every edge joining two vertices is replaced by an edge with the length of the shortest edge joining the two vertices, we know that

$$l(u_j \rightarrow u_{j+1})_{ISG} \leq l(u_j \rightarrow u_{j+1})_G \text{ with } 0 < j < n.$$

So,

$$l(Sol2_{ISG}) \leq l(Sol2)$$

But then

$$l(Sol2_{ISG}) \leq l(Sol2) = (1 + r) * l(IS) < (1 + q) * l(IS) \leq (1+p) * l(IS) = L_{limit}.$$

So, the length of the corresponding path in ISG is less than L_{limit}. Therefore all the vertices on the path $Sol2_{ISG}$ (and thus on the path *Sol2*), including u_i, should have been in G', contradicting our assumption.

If $q > p$, then G' may have been too "small", and a better solution may exist outside G', although this is not necessarily the case. Think of the case where we take $p = 0$, and keep in G' only the vertices on paths representing the Idealized Solutions in ISG. The length of the *actual* solutions in G along these paths will almost surely be larger than $l(IS)$.

8.3.7. Repair: second pass

In the last case, it is sufficient to set p to q in step (3) and calculate a new, larger reduced graph G''. An optimal solution in G'' is now guaranteed to be also an optimal solution in G, and therefore at worst we only need to cycle once.

Let us justify the last statements: in G'' we have all the vertices on paths representing Idealized Solutions (i.e. paths in ISG) of length $(l + q) * l(IS)$ or less. Assume that the optimal solution in G is:

$$Opt.Sol. : s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \ldots v_{m-1} \rightarrow v_m = t.$$

Its length must be at most the length of (the previously found) *Tent.Sol.* (which is also a solution in G), which is $(1 + q) * l(IS)$. The length of the corresponding path in ISG must be at most the "real" length (the length of the path in G), which is $(1 + q) * l(IS)$, and by construction all the vertices $v_0, v_2, \ldots, v_m$ are in G''. So, the value q_{new} in the second cycle must be at most q and the algorithm will stop.

8.4. The choice of the coefficient p

Let us consider the coefficient p in step (3) of SRM. In order to choose a value for p, we could use some heuristic taking into account some knowledge we have about the network. For instance we might know that the length of a real solution will be at most 140 percent of the length of its idealized solution, so we could set p to 0.40.

If p is chosen too small, SRM will almost surely cycle. This single repetition may prove costly: if q is large, i.e. if the Idealized Solution is very much better than the "real" solution in G', the graph G'' may not be much smaller than G. When p is chosen too large, the reduced set of vertices V' will be large and little benefit will accrue from the use of SRM.

It is not necessarily optimal to choose p so large that the single cycling of SRM *never* occurs. A smaller value of p could prevent cycling in most cases, and in these cases the search space could be considerably reduced. With such a smaller value of p, SRM will cycle only in some of the more extreme problems. Somewhere between a (too) large p and a (too) small p lies an optimal p, which depends on the graph, the actual cost of searching it, and the distribution of search requests. Such an optimal p can be determined empirically, after gathering enough information about a particular application, and assuming that future usage of the system will be essentially similar to past usage.

Another approach is to set p to 0. This way only the vertices on the path of the Idealized Solution will be in G'. When we search G', we shall only find solutions along the idealized path. Effectively, we traverse the idealized path (from ISG) in G. The cost of traversing this path only will obviously be small, however, the resulting solution may be bad. It will be very probable that in step (6) $q > p$. Only if the Idealized Solution is totally realizable in G will $p = q$. In the second cycle p will have the value of a real solution along the idealized path. The advantage will be a smaller search space (a smaller G') if the real optimal solution is likely to be along the idealized path. The value of p in the second pass is determined using knowledge about a route which has some chance of being near to optimal. The disadvantage of this approach is that we shall almost always have to cycle once.

8.5. Application

In an application, the Idealized Skeleton Graph need to be built only once for a given network, independently of the problem at hand. SRM reduces the size of the graph to be searched specifically for each problem, and may result in an increased performance. SRM appears particularly useful when searching large graphs where the search for a solution can often be confined "near" the Idealized Solution, i.e. much of the graph need never be searched. The benefits of SRM increase if a network is searched several times, because not only optimal but also near optimal solutions are of interest, or because there are several criteria for optimality. Since SRM reduces the size of the space to be searched, each of the solutions of interest will be found more efficiently than if the entire initial search space had been searched. The sum of the savings brought when finding each solution of interest may be very much larger than the cost of applying SRM. Results are given in chapter 14.

9. Heuristic Search

In order to further improve search efficiency we can make use of heuristics. One way to build an heuristic search algorithm is to make use of an heuristic function which gives for every vertex an estimate of the distance remaining to the goal vertex. A well-known algorithm for finding a minimum length path using heuristic estimates is A* (see [Ha, 1968]). Some generalizations of this algorithm have been elaborated in [Po, 1970], [Ha, 1974], [Me, 1984] and [Pe, 1979].

9.1. The A* search algorithm

We shall first give a definition of the A* search algorithm. We shall use a notation similar to the notation we used previously.

Consider a finite directed graph (V, E), where V is the set of vertices and E the set of directed edges joining two vertices from V. Each edge e from E has a length $l(e) \geq 0$. The two special vertices s and t of the graph are the *starting* and *terminating* vertices. We want to find a shortest directed path from s to t, where the length of a path is the sum of the lengths of its edges. The evaluation function $f(v)$ is an estimate of the length of a minimum length path constrained to go through the vertex v. The function $f(v)$ is composed of two parts: $\lambda(v)$ and $h(v)$. $\lambda(v)$, the label of the vertex v, is the length of the best known path from s to v. $h(v)$, the heuristic, is an estimate of the length of a minimum path from v to the goal vertex; we assume that $h(t) = 0$ for all heuristic functions h. The value of $\lambda(v)$ is determined during the propagation of the algorithm. The value of $h(v)$ is given initially for every vertex. The evaluation function $f(v)$ is composed as follows:

$$f(v) = \lambda(v) + h(v).$$

We now give the A* search algorithm.

(1) $\lambda(s) \leftarrow 0$ and $f(s) \leftarrow h(s)$. For all $v \in V$, $v \neq s$, $\lambda(v) \leftarrow \infty$.
(2) $F \leftarrow \{s\}$.

(3) If F is empty then stop, no path could be found. Otherwise, let u be a vertex in F for which $f(u)$ is minimum.
(4) If $u = t$, stop, a path is found.
(5) For every edge $e: u \rightarrow v$,
if $\lambda(v) > \lambda(u) + l(e)$,
then $\lambda(v) \leftarrow \lambda(u) + l(e)$ and $f(v) \leftarrow \lambda(v) + h(v)$ and $F \leftarrow F + \{v\}$.
(6) $F \leftarrow F - \{u\}$ and go to step (3).

If in step (3), there are ties, then it makes sense to work first on the vertex which is estimated closest to the goal vertex. So, the vertex with the smallest value for $h(v)$ should be chosen. Note that this way, the goal vertex, which obviously has an estimate of 0, will always be favoured in case of ties.

Note that the only difference between the A* algorithm, and the improved Dijkstra algorithm described in chapter 3, is the introduction of the heuristic estimate $h(v)$ and the absence of the collection T. The collection T cannot be used (in step (5)), because in the basic A* algorithm, a vertex may become the branching vertex u in step (3) multiple times, whereas in Dijkstra's algorithm this happened at most once for every vertex. We shall later see under which condition this can be avoided.

9.2. Admissibility of A*

It has been proven (see [Ha, 1968] or [Ni, 1980]) that if for every vertex v, $h(v)$ is a lower bound (an underestimate) of the actual length of a minimum path from v to the goal vertex, then the algorithm A* is admissible, i.e. it always finds an optimal path. This is easily seen by the fact that the real length of a path cannot be less than an underestimate of its length. Once a complete path has been found in step (4), the length of this path is real, it contains no estimate. So, if all other (incomplete) paths have an *underestimated* length which is higher than the actual length of the (complete) path we have found, then none of the completed paths developed out of the incomplete paths, can have an actual length which is lower.

9.3. Consistent heuristics

Earlier we said that a vertex could become the branching vertex multiple times in the A* algorithm. For an example, consider the graph in example 9.1. In this example the estimates (the h value of a vertex) are encircled. By inspection, we note that in this example, all estimates are in fact underestimates. Suppose we want to find a minimum length path from s to t.

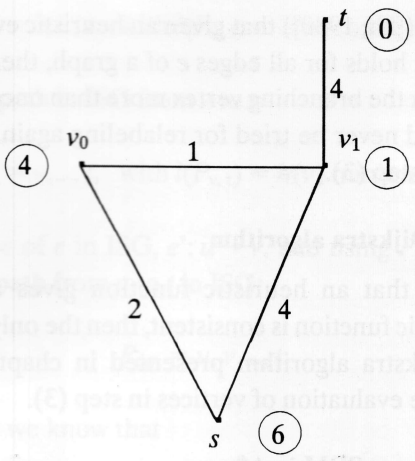


Fig. 9.1.

The first vertex to become the branching vertex is s. From s, v_1 is labeled 4 and $f(v_1)$ becomes 5. $\lambda(v_0)$ becomes 2 and $f(v_0)$ becomes 6. Then v_1 becomes the branching vertex since it has the lowest evaluation value. t is labeled 8 and $f(t)$ becomes 8. v_0 does not get relabeled: $h(v_0) + \lambda(v_0)$ via v_1 is $4 + 5 = 9$, which is higher than the previous value of 6. Then v_0 becomes the branching vertex. From v_0, v_1 does get relabeled to 3 (even though it had been the branching vertex once already) and its evaluation value becomes 4. Consequently v_1 is put in F again and it becomes the next branching vertex. So, even though the heuristic estimates are underestimates, a vertex can become the branching vertex multiple times.

A vertex can become the branching vertex multiple times when the heuristic evaluation function (the function which is composed of the label and the heuristic estimate) is not monotonically non-decreasing along the path from s to t. If an heuristic evaluation function is monotonically non-decreasing along any path in the graph, then the heuristic is called *monotone* or *consistent* (see [Ni, 1980]). A monotone heuristic evaluation function translates into the following *consistency assumption*:

Consider an edge $e : u \rightarrow v$, then

$$h(u) - h(v) \leq l(e).$$

In example 9.1, the consistency assumption is not valid for the edge $s \rightarrow v_1$ and for the edge $v_0 \rightarrow v_1$. The estimate for vertex v_1 is not consistent with the estimates of s and v_0, considering the lengths of the edges $s \rightarrow v_1$ and $v_0 \rightarrow v_1$ respectively.

It has been proven ([Ni, 1980]) that given an heuristic evaluation function, if the consistency assumption holds for all edges e of a graph, then A* using this function will never make a vertex the branching vertex more than once. A vertex that has been a branching vertex need never be tried for relabeling again. The label of the vertex becomes *permanent* in step (3).

9.4. A* as a modified Dijkstra algorithm

If we make sure that an heuristic function gives an underestimate, and moreover, if the heuristic function is consistent, then the only difference between A* and the improved Dijkstra algorithm presented in chapter 3, is the use of the heuristic function in the evaluation of vertices in step (3).

9.5. Using the results from SRM in A*

As a result of the determination of the search space in SRM (the determination of the graph G'), for each vertex v in the search space we have an estimate of its distance from the source vertex ($\delta_s(v)$), and an estimate of the distance remaining to the goal vertex ($\delta_g(v)$). If we make sure that these estimates are underestimates, and that these estimates are consistent, then we may use these estimates in the evaluation step of the algorithms to search discrete and discrete dynamic networks (see chapters 4 and 6).

It is easily seen that the estimates obtained from SRM are underestimates by the fact that in the Idealized Skeleton Graph (see chapter 8), all parallel edges are replaced by one edge with a length corresponding to the length of the shortest of the parallel edges. So, when using one of the parallel edges in the real graph, its length is at least the length of its representative edge in ISG. Since the estimates are determined by searching for optimal paths in ISG, these estimates must be underestimates of the real optimal paths.

The fact that the estimates from SRM are consistent is proven by contradiction. Suppose that for some edge $e: u \rightarrow v$ ($e \in E'$), with u and v part of the search space (the reduced graph), the consistency assumption does not hold, i.e. :

$$h(u) - h(v) > l(e) \qquad (1).$$

Where $h(u)$ and $h(v)$ are the estimates of the distance remaining to the goal vertex t from u and v respectively. These estimates were found by searching ISG for optimal paths ($\delta_s(u)$, $\delta_g(u)$). So, we know that there exists a path in ISG from u to t:

$$P_{u,t} = u,..., t, \quad \text{with } l(P_{u,t}) = h(u) \text{ minimum.}$$

Similarly, there exists a path in ISG from v to t:

$$P_{v,t} = v,..., t, \quad \text{with } l(P_{v,t}) = h(v) \text{ minimum.}$$

Using the representative of e in ISG, e': $u \rightarrow v$, and using $P_{v,t}$ it is then possible to construct the following path from u to t in ISG:

$$P_{opt} = u, v,..., t.$$

By the definition of ISG we know that

$$l(e') \leq l(e) \qquad (2).$$

Obviously,

$$l(P_{opt}) = l(P_{v,t}) + l(e').$$

By (2) and since $l(P_{v,t}) = h(v)$ we get:

$$l(P_{opt}) = h(v) + l(e') \leq h(v) + l(e).$$

By (1) we know that

$$h(v) + l(e) < h(u).$$

But then

$$l(P_{opt}) \leq h(v) + l(e) < h(u).$$

So, we have constructed a path from u to t in ISG for which the length is less that $h(u)$. Since $h(u)$ is minimum, this is a contradiction. So it must be that $h(u) - h(v) \leq l(e)$.

9.6. Using heuristics in DYNET: DYNET*

Now that we have shown that the estimates that we get from SRM are consistent and in fact underestimates, we can easily adapt our DYNET algorithm for searching a discrete dynamic network to include heuristic estimates. All we need to do is to

change the evaluation in step (3). Furthermore, only the vertices that were determined to be in the search space using SRM, are considered in the search. For a detailed description of the other steps of the algorithm, see chapter 6. We shall refer to this algorithm as the DYNET* algorithm.

Consider a discrete dynamic network consisting of the graph $G = (V, E)$ and the connection cost function *CON*. The maximum value of *CON* at a vertex u is *maxiCON(u)*, which is non-zero. The two special vertices s and t of the network are the *starting* and *terminating* vertices. We want to find a legal path from s to t in our discrete dynamic network, where the end value of the path is minimum and given this end value, the start value of the path is maximum and at least T_{start}.

Let the collection V' be the collection of vertices that were determined to be in the search space using SRM (the reduced set of vertices). Furthermore let $\delta_s(v)$ denote the estimate of the distance from s to v, determined using SRM, and let $\delta_t(v)$ denote the estimate of the distance from v to t.

Pass 1:

(1) $\lambda(s) \leftarrow T_{start}$ and for all $v \in V'$, $v \neq s$, $\lambda(v) \leftarrow \infty$.
Create a partial path P_0 consisting of s only, $end(P_0) \leftarrow T_{start}$.
For all $v \in V'$, $\omega(v, u) = \infty$ for each neighbour u of v, $u \in V'$.
(2) $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j$ in F for which $end(P_m) + \delta_t(u_j)$ is minimum; if F is empty then stop, no complete path could be found.
(4) If $u_j = t$, stop, P_m is a complete path with an optimal end value.
(5) If $end(P_m) < \lambda(u_j) + maxiCON(u_j)$.
then for every relevant edge $e_j : u_j \rightarrow u_{j+1}; u_{j+1} \in V'$:
if $\lambda(u_{j+1}) > end(e_j)$
then $\lambda(u_{j+1}) \leftarrow end(e_j)$
if $end(e_j) < \lambda(u_{j+1}) + maxiCON(u_{j+1})$
then create a partial path $P_n = s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j, e_j, u_{j+1}$ and
$F \leftarrow F + \{P_n\}$.
if $end(e_j) < \omega(u_{j+1}, u_j)$
then $\omega(u_{j+1}, u_j) \leftarrow end(e_j)$.
(6) $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_0, \ldots, u_{j-1}, e_{j-1}, u_j$ and a vertex u_{j+1}, then the relevant edges from u_j to u_{j+1} are the edges e_j: $u_j \rightarrow u_{j+1}$ for which the following two *ordered* conditions hold:

(1) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$,
(2) $end(e_j) < end(e_{min}) + maxiCON(u_{j+1})$,
where $end(e_{min})$ is the minimum end value of any edge satisfying (1).

Pass 2:

(1) $\kappa(t) \leftarrow \lambda(t)$ and for all $v \in V$, $v \neq t$, $\kappa(v) \leftarrow -\infty$.
Create a partial path P_0 consisting of t only, $start(P_0) \leftarrow \lambda(t)$.
(2) $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $u_j, \ldots, u_{k-1}, e_{k-1}, t$ in F for which $start(P_m) - \delta_s(u_j)$ is maximum.
(4) If $u_j = s$, stop, P_m is an optimal complete path.
(5) If $start(P_m) > \kappa(u_j) - maxiCON(u_j)$
then for every relevant edge e_{j-1} : $u_{j-1} \rightarrow u_j$;
with $u_{j-1} \in V$ and $\omega(u_j, u_{j-1}) \leq start(P_m)$:
if $\kappa(u_{j-1}) < start(e_{j-1})$
then $\kappa(u_{j-1}) \leftarrow start(e_{j-1})$
if $start(e_{j-1}) > \kappa(u_{j-1}) - maxiCON(u_{j-1})$
then create a partial path $P_n = u_{j-1}, e_{j-1}, u_j, \ldots, u_{k-1}, e_{k-1}, t$ and
$F \leftarrow F + \{P_n\}$.
(6) $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_j, e_j, u_{j+1}, \ldots, u_k$ and a vertex u_{j-1}, then the relevant edges from u_{j-1} to u_j are the edges e_{j-1}: $u_{j-1} \rightarrow u_j$ for which the following two *ordered* conditions hold:

(1) $end(e_{j-1}) \leq start(e_j) - CON(u_j, e_{j-1}, e_j)$,
(2) $start(e_{j-1}) > start(e_{max}) - maxiCON(u_{j-1})$,
where $start(e_{max})$ is the maximum start value of any edge satisfying (1).

10. Offset Vertices

An excellent way to decrease the amount of search necessary to find an optimal path in a graph, is to make sure that the graph that is being searched is as small as possible. In a graph, some categories of vertices need not be considered during search. In this chapter we describe one such category: the offset vertices. An offset vertex is a vertex which lies between exactly two other vertices. A path leading to such a vertex can only be continued to one vertex, and consequently the vertex does not need to be considered during search if it is neither the starting vertex nor the terminating vertex. The fewer vertices need to be considered during search, the faster the search will be. We also describe how offset vertices can be determined in (directed) graphs and in discrete dynamic networks. Furthermore we describe adjustments to the search strategies for these graphs, in case the starting or terminating vertex is an offset vertex.

10.1. Offset vertices

Let us consider the graph in fig. 10.1. If we are searching this graph using, say, Dijkstra's algorithm, then whenever we arrive at vertex v_2, which lies between exactly two other vertices (v_1 and v_3), we can only continue our path by going to v_3 if we arrived at v_2 from v_1, or to vertex v_1 if we came from v_3. There is no other choice since continuing the path back to the vertex we came from is obviously useless when we are searching for an optimal path. Unless v_2 is the starting vertex or the terminating vertex, as soon as we are going from v_1 to v_2, adding a length of 2, we know for sure that from v_2 we shall continue to v_3, adding a length of 3, giving a total of 5 to reach v_3 from v_1. Similarly, if we are going from v_3 to v_2, adding a length of 3, we know for sure that from v_2 we shall continue to v_1, adding a length of 2, giving a total of 5 to reach v_1 from v_3. So, if v_2 is neither the starting vertex nor the terminating vertex, when searching the graph of fig. 10.1, we can equivalently search the graph of fig. 10.2, which is smaller. The vertex v_2 has been removed and has become an *offset vertex*. In fig. 10.2, if v_2 is the starting vertex or the terminating vertex, we need to remember that v_2 (the offset vertex) is lying between v_1 and v_3 (the node vertices),

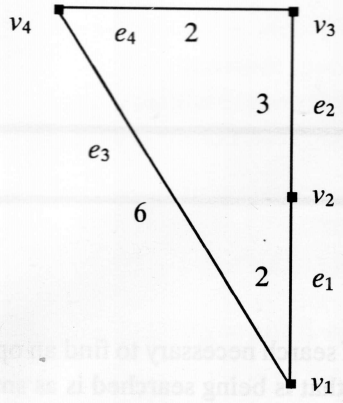


Fig. 10.1.

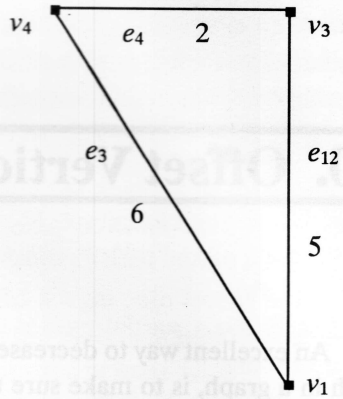


Fig. 10.2.

separated by a length of 2 from v_1 and by a length of 3 from v_3 (the distance offsets). We shall now define the concepts of offset vertices and distance offsets more precisely.

10.2. Offset vertices in undirected graphs

We can include offset vertices in the definition of a (weighted), undirected graph as follows. A graph G with offset vertices is a structure which consists of three sets and a function:

(1) a set of node vertices V_{node},
(2) a set of offset vertices V_{off},
(3) a set of (undirected) edges E,
(4) a non-negative, real-valued function, called the distance offset function *DIS*, having two vertices as arguments: one offset vertex and one node vertex.

Each edge e from E is incident to the elements of an unordered pair of node vertices. Each edge e is assigned a non-negative length $l(e)$. With each offset vertex $v \in V_{\text{off}}$, we associate 2 values, $\gamma_1(v)$ and $\gamma_2(v)$, which denote the node vertices connected to the offset vertex. The distance offset function *DIS* specifies the length separating an offset city and the node cities connected to it.

In the example of fig. 10.2 $V_{\text{node}} = \{v_1, v_3, v_4\}$, $V_{\text{off}} = \{v_2\}$, and $E = \{e_{12}, e_3, e_4\}$. Furthermore:

$$\gamma_1(v_2) = v_1,$$
$$\gamma_2(v_2) = v_3,$$

and

$$DIS(v_2, v_1) = DIS(v_1, v_2) = 2,$$
$$DIS(v_2, v_3) = DIS(v_3, v_2) = 3.$$

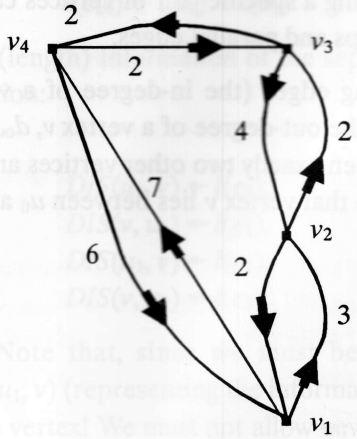


Fig. 10.3.

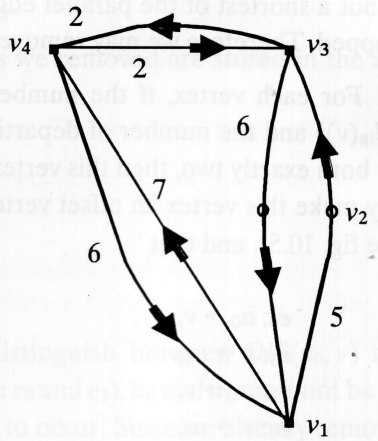


Fig. 10.4.

10.3. Offset vertices in directed graphs

In a directed graph, the length separating two vertices may depend on the direction. For example, in fig. 10.3, v_2 lies between exactly two other vertices, v_1 and v_3, and thus can be made an offset vertex. The length of the edge from v_1 to v_2 is 3, whereas the length of the edge from v_2 to v_1 is 2. So, when we introduce offset vertices to directed (weighted) graphs, the order of the arguments of the *DIS* function becomes important. In fig. 10.4 v_2 has been made an offset vertex. The DIS function becomes:

$$DIS(v_1, v_2) = 3,$$
$$DIS(v_2, v_1) = 2,$$
$$DIS(v_3, v_2) = 4,$$
$$DIS(v_2, v_3) = 2.$$

10.3.1. Transforming a graph

In this paragraph we describe the steps to transform a directed, weighted graph into a graph with offset cities, which can be equivalently, but more efficiently searched for an optimal path.

First we make the observation that when we are searching for an optimal path in a graph, it makes no sense to have self-loops and parallel edges. A self-loop takes us back to the same vertex we came from at an extra cost, which will not give an optimal path. When parallel edges exist between two vertices, for an optimal path only a shortest edge of the parallel edges will be used to go from one vertex to the other, so all but a shortest of the parallel edges connecting a specific pair of vertices can be dropped. Therefore we may remove all self-loops and parallel edges.

For each vertex, if the number of arriving edges (the in-degree of a vertex v, $d_{in}(v)$) and the number of departing edges (the out-degree of a vertex v, $d_{out}(v)$), are both exactly two, then this vertex lies between exactly two other vertices and we may make this vertex an offset vertex. Suppose that vertex v lies between u_0 and u_1 (see fig. 10.5), and that

$$e_0\colon u_0 \to v,$$
$$e_1\colon v \to u_1,$$
$$e_2\colon u_1 \to v,$$
$$e_3\colon v \to u_0.$$

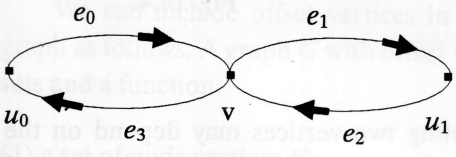


Fig. 10.5.

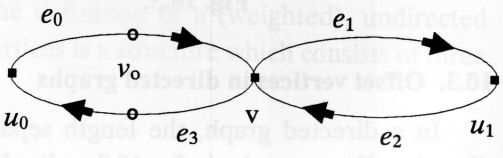


Fig. 10.6.

When we make v an offset vertex, u_0 and u_1 become the node vertices which are connected to the offset vertex:

$$\gamma_1(v) \leftarrow u_0,$$
$$\gamma_2(v) \leftarrow u_1.$$

We remove e_0 and e_1 from the graph and replace them by one compound edge from u_0 to u_1. The length of this new edge becomes the sum of the lengths of e_0 and e_1:

$$e_{01}\colon u_0 \to u_1,$$
$$l(e_{01}) \leftarrow l(e_0) + l(e_1).$$

Similarly, we remove e_2 and e_3 from the graph and replace them by one compound edge from u_1 to u_0. The length of this new edge becomes the sum of the lengths of e_2 and e_3:

$$e_{10}: u_1 \rightarrow u_0,$$
$$l(e_{10}) \leftarrow l(e_2) + l(e_3).$$

The (length) information of the separate edges we removed are stored in the *DIS* function:

$$DIS(u_0, v) \leftarrow l(e_0),$$
$$DIS(v, u_1) \leftarrow l(e_1),$$
$$DIS(u_1, v) \leftarrow l(e_2),$$
$$DIS(v, u_0) \leftarrow l(e_3).$$

Note that, since we must be able to distinguish between $DIS(u_0, v)$ and $DIS(u_1, v)$ (representing the information of edge e_0 and e_2), u_0 and u_1 must not be the same vertex! We must not allow new self-loops to occur. Since we already removed self-loops, we know for sure that v will never be the same vertex as u_0 or u_1 if we do not allow new self-loops to occur.

It may be, that the vertex v that we have made an offset vertex had already been made a node vertex for some other offset vertex v_o. Since an offset vertex is connected to exactly two node vertices, and since v is connected to exactly two node vertices, v_o must lie either between u_0 and v, or between v and u_1. So, either

$$\gamma_1(v_o) = u_0 \text{ and } \gamma_2(v_o) = v,$$

or

$$\gamma_1(v_o) = u_1 \text{ and } \gamma_2(v_0) = v.$$

In the first case (see fig. 10.6), u_1 replaces v as the node city which is connected to v_o:

$$\gamma_2(v_o) \leftarrow u_1.$$

In the second case u_0 replaces v as the node city which is connected to v_o:

$$\gamma_2(v_o) \leftarrow u_0.$$

Furthermore we need to change the *DIS* function to give the distances from v_0 to the new node city to which it is connected instead of v. For the first case:

$$DIS(v_o, u_1) \leftarrow DIS(v_o, v) + DIS(v, u_1),$$
$$DIS(u_1, v_o) \leftarrow DIS(u_1, v) + DIS(v, v_o).$$

For the second case:

$$DIS(v_o, u_0) \leftarrow DIS(v_o, v) + DIS(v, u_0),$$
$$DIS(u_0, v_o) \leftarrow DIS(u_0, v) + DIS(v, v_o).$$

10.3.2. The algorithm to transform a directed, weighted graph

We now give a formal definition of the algorithm to transform a directed, weighted graph into a directed, weighted graph with offset cities:

Consider a finite, directed, weighted graph $G = (V, E)$. We shall construct an equivalent graph with offset vertices $G' = (V_{node}, V_{off}, E', DIS)$, where V_{node} is the collection of node vertices, V_{off} the collection of offset vertices, E' the collection of directed edges joining two vertices from V_{node}, and DIS is the distance offset function having two arguments: a node vertex and an offset vertex. Each edge e from E' has a length $l(e) \geq 0$. Furthermore, with each offset vertex $v \in V_{off}$, we will associate 2 values, $\gamma_1(v)$ and $\gamma_2(v)$, which denote the node vertices which are connected to the offset vertex.

(1) $V_{off} \leftarrow \varnothing$, $V_{node} \leftarrow \varnothing$.
(2) $E' \leftarrow E$. Remove from E' all self-loops, and all parallel edges but the shortest ones.
(3) For each vertex $v \in V$,
if $d_{in}(v) = d_{out}(v) = 2$ then
Suppose
e_0: $u_0 \rightarrow v$,
e_1: $v \rightarrow u_1$,
e_2: $u_1 \rightarrow v$,
e_3: $v \rightarrow u_0$.
if $u_0 \neq u_1$
$\gamma_1(v) \leftarrow u_0$ and $\gamma_2(v) \leftarrow u_1$.
$V_{off} \leftarrow V_{off} + \{ v \}$.
$V_{node} \leftarrow V_{node} + \{ u_0, u_1 \}$.
$E' \leftarrow E' - \{ e_0, e_1, e_2, e_3 \}$.
Create an edge e_{01}: $u_0 \rightarrow u_1$, $l(e_{01}) \leftarrow l(e_0) + l(e_1)$.
Create an edge e_{10}: $u_1 \rightarrow u_0$, $l(e_{10}) \leftarrow l(e_2) + l(e_3)$.

$E' \leftarrow E' + \{ e_{01}, e_{10} \}$.
$DIS(u_0, v) \leftarrow l(e_0)$,
$DIS(v, u_1) \leftarrow l(e_1)$,
$DIS(u_1, v) \leftarrow l(e_2)$,
$DIS(v, u_0) \leftarrow l(e_3)$.
if $v \in V_{node}$ then there exist offset vertices v_o for which v was a node vertex. For each such offset vertex v_o:
if $\gamma_1(v_o) = u_0$ and $\gamma_2(v_o) = v$.
then $\gamma_2(v_o) \leftarrow u_1$,
$DIS(v_o, u_1) \leftarrow DIS(v_o, v) + DIS(v, u_1)$,
$DIS(u_1, v_o) \leftarrow DIS(u_1, v) + DIS(v, v_o)$.
else $\gamma_2(v_o) \leftarrow u_0$,
$DIS(v_o, u_0) \leftarrow DIS(v_o, v) + DIS(v, u_0)$,
$DIS(u_0, v_o) \leftarrow DIS(u_0, v) + DIS(v, v_o)$.
$V_{node} \leftarrow V_{node} - \{ v \}$.
else $V_{node} \leftarrow V_{node} + \{ v \}$.
else $V_{node} \leftarrow V_{node} + \{ v \}$.

10.3.3. Adapting Dijkstra's algorithm to offset vertices

Now that we have defined a graph with offset vertices, and an algorithm to transform a weighted, directed graph into a graph with offset vertices which can be searched more efficiently for an optimal path, we adapt a graph search algorithm to handle such a graph with offset vertices. As a basis we use the improved version of Dijkstra's algorithm as defined in chapter 3.

As we saw in section 10.1, we only need to consider offset vertices during search if an offset vertex is either the starting vertex or the terminating vertex. We shall look at each case in turn.

10.3.3.1. An offset vertex as the starting vertex

In the definition of the improved Dijkstra algorithm (see section 3.1.2.3.) the starting vertex s is handled in step (1): the starting vertex is labeled with 0 and is put in the frontier. Since it is the only vertex that is put in the frontier, at step (3) the starting vertex is selected as the branching vertex and in step (5) all its neighbours are visited (labeled and put in the frontier). Finally, the starting vertex is removed from the frontier. If the starting vertex is an offset vertex, we do not label it, but instead, the node vertices adjacent to the offset vertex are labeled with their respective distance from the offset vertex. The information of the edges departing

from the offset vertex can be recovered by using the γ values of the offset vertex to determine to which node cities it is connected, and by using the *DIS* function to determine the length of the connecting edges:

$$\lambda(\gamma_1(s)) \leftarrow DIS(s, \gamma_1(s)),$$
$$\lambda(\gamma_2(s)) \leftarrow DIS(s, \gamma_2(s)).$$

Since a vertex is always removed from the frontier as soon as all of its neighbours have been visited, we do not bother to actually put the source vertex itself in the frontier. Instead, after we have labeled its adjacent node vertices, we put those in the frontier. The offset vertex is temporarily treated as a node vertex by visiting its adjacent node vertices.

10.3.3.2. An offset vertex as the terminating vertex

When the terminating vertex t is an offset vertex, it is treated as a node vertex which is being labeled when one of its adjacent node vertices becomes permanent. We can recognize a node vertex u adjacent to the offset vertex by comparing it to the γ values of the offset vertex. We can recover the information about the edge connecting the node vertex and the offset vertex by using the *DIS* function. With this information we are able to label the offset vertex:

if $u = \gamma_1(t)$ or $u = \gamma_2(t)$:
  if $\lambda(t) > \lambda(u) + DIS(u, t)$
  then $\lambda(t) \leftarrow \lambda(u) + DIS(u, t)$

We do actually have to treat the offset vertex as a node vertex and put it in the frontier. If we would follow the same scheme as when the starting vertex is an offset vertex, i.e. label the node vertices with the distance from its neighbours *plus* the distance the offset vertex is from the node vertex, then we would get an incorrect answer in a case like fig. 10.7. From s we label u_0 with 5. If we would label u_0 with $5 + 4 = 9$ and stop when it becomes the branching vertex the next iteration, we would miss the shorter path via u_1. Because we label u_0 with a distance greater than the actual distance, there may be neighbours of u_0 which have a smaller actual label than u_0, and could result in a shorter route, if we would label them.

10.3.3.3. An offset vertex as starting vertex and terminating vertex

When both the starting vertex and the terminating vertex are offset vertices, a special case arises when they are both adjacent to the same pair of node vertices. This case must be handled in a separate step. The distance separating the two offset

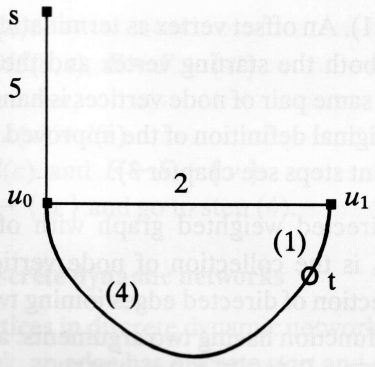


Fig. 10.7.

vertices must be calculated by going from one offset vertex to the other directly, without passing a node vertex. The different cases are shown in fig. 10.8:

$\gamma_1(s) = \gamma_1(t)$, and $\gamma_2(s) = \gamma_2(t)$
$DIS(\gamma_1(s), s) < DIS(\gamma_1(t), t)$
$\lambda(t) \leftarrow DIS(\gamma_1(t), t) - DIS(\gamma_1(s), s)$

$\gamma_1(s) = \gamma_1(t)$, and $\gamma_2(s) = \gamma_2(t)$
$DIS(\gamma_1(s), s) > DIS(\gamma_1(t), t)$
$\lambda(t) \leftarrow DIS(\gamma_2(t), t) - DIS(\gamma_2(s), s)$

$\gamma_1(s) = \gamma_2(t)$, and $\gamma_2(s) = \gamma_1(t)$
$DIS(\gamma_1(s), s) < DIS(\gamma_2(t), t)$
$\lambda(t) \leftarrow DIS(\gamma_2(t), t) - DIS(\gamma_1(s), s)$

$\gamma_1(s) = \gamma_2(t)$, and $\gamma_2(s) = \gamma_1(t)$
$DIS(\gamma_1(s), s) > DIS(\gamma_2(t), t)$
$\lambda(t) \leftarrow DIS(\gamma_1(t), t) - DIS(\gamma_2(s), s)$

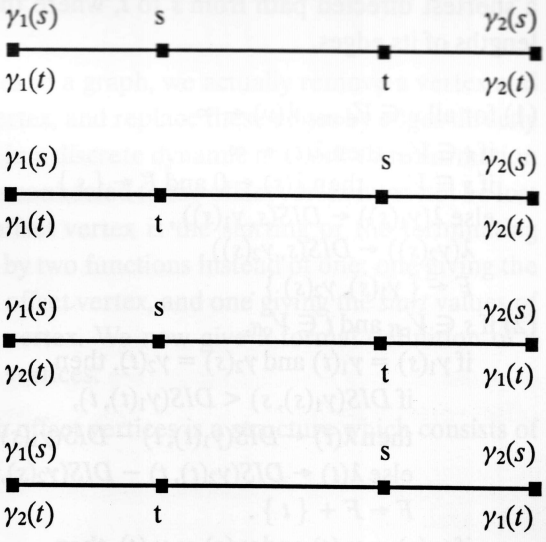


Fig. 10.8.

We must handle this special case as above. If we calculated the distance by first going from one offset vertex to one of its adjacent node vertices, and then from the node vertex to the other offset vertex for example, then the solution would not be optimal.

10.3.4. The Dijkstra algorithm for searching a graph with offset vertices

We now give a formal definition of Dijkstra's algorithm, adapted to handle an offset vertex as starting vertex or terminating vertex. An offset vertex as starting

vertex is handled in step (1). An offset vertex as terminating vertex is handled in step (6). The special case of both the starting vertex and the terminating vertex as an offset vertex between the same pair of node vertices is handled in step (2). The other steps are similar to the original definition of the improved Dijkstra algorithm (for an explanation of the different steps see chapter 3).

Consider a finite directed weighted graph with offset vertices, $G = (V_{node}, V_{off}, E, DIS)$, where V_{node} is the collection of node vertices, V_{off} the collection of offset vertices, E the collection of directed edges joining two vertices from V_{node}, and DIS is the distance offset function having two arguments: a node vertex and an offset vertex. Each edge e from E has a length $l(e) \geq 0$. With each offset vertex $v \in V_{off}$, we associate 2 values, $\gamma_1(v)$ and $\gamma_2(v)$, which denote the node vertices the offset vertex is connected to. The two special vertices s and t of the graph are the *starting* and *terminating* vertices (each vertex either a node or an offset vertex). We want to find a shortest directed path from s to t, where the length of a path is the sum of the lengths of its edges.

(1) for all $v \in V_{node}$, $\lambda(v) \leftarrow \infty$.
  if $t \in V_{off}$ then $\lambda(t) \leftarrow \infty$.
  if $s \in V_{node}$ then $\lambda(s) \leftarrow 0$ and $F \leftarrow \{ s \}$.
  else $\lambda(\gamma_1(s)) \leftarrow DIS(s, \gamma_1(s))$,
    $\lambda(\gamma_2(s)) \leftarrow DIS(s, \gamma_2(s))$,
    $F \leftarrow \{ \gamma_1(s), \gamma_2(s) \}$.
(2) if $s \in V_{off}$ and $t \in V_{off}$,
  if $\gamma_1(s) = \gamma_1(t)$ and $\gamma_2(s) = \gamma_2(t)$, then
    if $DIS(\gamma_1(s), s) < DIS(\gamma_1(t), t)$,
    then $\lambda(t) \leftarrow DIS(\gamma_1(t), t) - DIS(\gamma_1(s), s)$
    else $\lambda(t) \leftarrow DIS(\gamma_2(t), t) - DIS(\gamma_2(s), s)$
    $F \leftarrow F + \{ t \}$.
  if $\gamma_1(s) = \gamma_2(t)$ and $\gamma_2(s) = \gamma_1(t)$, then
    if $DIS(\gamma_1(s), s) < DIS(\gamma_2(t), t)$,
    then $\lambda(t) \leftarrow DIS(\gamma_2(t), t) - DIS(\gamma_1(s), s)$
    else $\lambda(t) \leftarrow DIS(\gamma_1(t), t) - DIS(\gamma_2(s), s)$
    $F \leftarrow F + \{ t \}$.
(3) $T \leftarrow V_{node}$.
(4) let u be a vertex in F for which $\lambda(u)$ is minimum; if F is empty then stop, no path could be found.
(5) if $u = t$, stop, an optimal path has been found.
(6) if $t \in V_{off}$, and $u = \gamma_1(t)$ or $u = \gamma_2(t)$:

if $\lambda(t) > \lambda(u) + DIS(u, t)$
then $\lambda(t) \leftarrow \lambda(u) + DIS(u, t)$, $F \leftarrow F + \{ t \}$.

(7) for each edge $e \in E$, $e: u \rightarrow v$,
if $v \in T$ and $\lambda(v) > \lambda(u) + l(e)$
then $\lambda(v) \leftarrow \lambda(u) + l(e)$ and $F \leftarrow F + \{ v \}$.

(8) $T \leftarrow T - \{ u \}$, $F \leftarrow F - \{ u \}$ and go to step (4).

10.4. Offset vertices in discrete dynamic networks

Introducing offset vertices in discrete dynamic networks is more complicated. In a discrete dynamic network, an edge has discrete *start* and *end* values. Parallel edges may have different *start* and *end* values. Even parallel edges with the same *start* and *end* values may have different *CON*nection characteristics. Therefore we cannot simply remove parallel edges. Since, as in a directed weighted graph, a self-loop takes us back to the vertex we came from at an extra cost, self-loops may always be removed.

When we introduce offset vertices in a graph, we actually remove a vertex and the edges connected to the (offset) vertex, and replace these edges by edges directly connecting the two node cities. Since in a discrete dynamic network the information of the *start* and *end* values of the edges connected to the offset vertex must not be lost (we need to know them when the offset vertex is the starting or the terminating vertex), we need to extend our graph by two functions instead of one: one giving the *end* values of the edges arriving at an offset vertex, and one giving the *start* values of the edges departing from the offset vertex. We now give a formal definition of a discrete dynamic network with offset vertices.

A discrete dynamic network with offset vertices is a structure which consists of three sets and three functions:

(1) a set of node vertices V_{node},
(2) a set of offset vertices V_{off},
(3) a set of edges E,
(4) a non-negative, real-valued function, called the connection function *CON*, having three arguments: a node vertex and two edges,
(5) a non-negative, real-valued function, called *START*, having three arguments: an offset vertex, a node vertex and an edge,
(6) a non-negative, real-valued function, called *END*, having three arguments: a node vertex, an offset vertex and an edge.

Each edge e from E is incident to the elements of an ordered pair of node vertices. With each edge e we associate two values: a start value *start(e)* and an end value *end(e)*. The connection function *CON* gives the required margin for a specific connection (for a more thorough discussion of a discrete dynamic network, see chapter 5). With each offset vertex $v \in V_{off}$, we associate 2 values, $\gamma_1(v)$ and $\gamma_2(v)$, which denote the node vertices the offset vertex is connected to. The *START* and *END* functions give the *start* and *end* values of the edges connected to offset vertices.

10.4.1. Transforming a discrete dynamic network

In this paragraph we describe steps to transform a discrete dynamic network into a discrete dynamic network with offset vertices, which can be equivalently, but more efficiently searched for an optimal path.

Since we cannot remove parallel edges, we cannot test the in- and out-degree to determine whether a vertex lies between exactly two other vertices. Instead, we explicitly test whether all edges which arrive at an offset vertex come from exactly one start (node) vertex, and have a connecting edge to a unique end (node) vertex, and whether all edges which depart from an offset vertex go to exactly one end (node) vertex, and have a preceding (connecting) edge from a unique start (node) vertex. In a railway services network, this translates to the condition that each train arriving at an offset station must come from a unique previous station, and there must exist a departing train to a unique next station, which forms a connection, and that each train departing from an offset station must go to a unique next station, and there must exist a train from a unique previous station, which forms a connection. Suppose vertex v is such a vertex (see fig. 10.5). Precisely, a vertex v can be made an offset vertex if we have:

if there exist two vertices u_0 and u_1, $u_0 \neq v$, $u_1 \neq v$, $u_0 \neq u_1$,
  such that
    for all edges e_k: $u_p \rightarrow v$ we have $u_p = u_0$ or $u_p = u_1$, and
    for all edges e_l: $v \rightarrow u_q$ we have $u_q = u_0$ or $u_q = u_1$, and
if each edge e_i: $u_0 \rightarrow v$, has a connecting edge e_{i+1}: $v \rightarrow u_1$, with $u_0 \neq u_1$,
and each edge e_{i+1}: $v \rightarrow u_1$, has a preceding edge e_i: $u_0 \rightarrow v$,
  such that
    $start(e_{i+1}) \geq end(e_i) + CON(v, e_i, e_{i+1})$,
and if each edge e_j: $u_1 \rightarrow v$, has a connecting edge e_{j+1}: $v \rightarrow u_0$,
and each edge e_{j+1}: $v \rightarrow u_0$, has a preceding edge e_j: $u_1 \rightarrow v$,
  such that
    $start(e_{j+1}) \geq end(e_j) + CON(v, e_j, e_{j+1})$,

We may then make v an offset vertex. The node vertices which are connected to v are u_0 and u_1:

$$\gamma_1(v) \leftarrow u_0,$$
$$\gamma_2(v) \leftarrow u_1.$$

Each pair of connecting edges e_i and e_{i+1}, connecting u_0 and v, and v and u_1 respectively, is removed from the graph and replaced by one compound edge e_{01} from u_0 to u_1, per connecting pair. This edge becomes the *start* value of e_i and the *end* value of e_{i+1}:

$$e_{01}: u_0 \rightarrow u_1,$$
$$start(e_{01}) \leftarrow start(e_i),$$
$$end(e_{01}) \leftarrow end(e_{i+1}).$$

The *end* value of e_i at v and the *start* value of e_{i+1} at v (which correspond to the arrival and departure at the offset vertex) are stored in the *START* and *END* functions, respectively:

$$END(u_0, v, e_{01}) \leftarrow end(e_i),$$
$$START(v, u_1, e_{01}) \leftarrow start(e_{i+1}).$$

All edges e which formed a connection with e_i at u_0 must also form a connection with e_{01} at u_0:

For each edge e for which $CON(u_0, e, e_i)$ is defined:
$$CON(u_0, e, e_{01}) \leftarrow CON(u_0, e, e_i).$$

All edges e which formed a connection with e_{i+1} at u_1 must also form a connection with e_{01} at u_1:

For each edge e for which $CON(u_1, e_{i+1}, e)$ is defined:
$$CON(u_1, e_{01}, e) \leftarrow CON(u_1, e_{i+1}, e).$$

Similarly, each pair of connecting edges e_j and e_{j+1}, connecting u_1 and v, and v and u_0 respectively, is removed from the graph and replaced by one compound edge e_{10} from u_1 to u_0, per connecting pair. This edge becomes the *start* value of e_j and the *end* value of e_{j+1}:

$$e_{10}: u_1 \rightarrow u_0,$$
$$start(e_{10}) \leftarrow start(e_j),$$

$$end(e_{10}) \leftarrow end(e_{j+1}).$$

The *end* value of e_j at v and the *start* value of e_{j+1} at v are stored in the *START* and *END* functions, respectively:

$$END(u_1, v, e_{10}) \leftarrow end(e_j),$$
$$START(v, u_0, e_{10}) \leftarrow start(e_{j+1}).$$

All edges e which formed a connection with e_j at u_1 must also form a connection with e_{10} at u_1:

For each edge e for which $CON(u_1, e, e_j)$ is defined:
$CON(u_1, e, e_{10}) \leftarrow CON(u_1, e, e_j)$.

All edges e which formed a connection with e_{j+1} at u_0 must also form a connection with e_{10} at u_0:

For each edge e for which $CON(u_0, e_{j+1}, e)$ is defined:
$CON(u_0, e_{10}, e) \leftarrow CON(u_0, e_{j+1}, e)$.

Note that since the *START* and *END* functions have an edge as third parameter, the two (node) vertex parameters can be allowed to be the same vertex. The edge parameter specifies the direction. So, we may allow new self-loops to occur. This means that we do have to make sure that neither u_0 nor u_1 is the same vertex as v (in this case v would become both an offset vertex *and* a node vertex at the same time, which cannot be allowed).

It may be, that the vertex v that we have made an offset vertex had already been made a node vertex for some other offset vertex v_o. Since an offset vertex is connected to exactly two node vertices, and since v is connected to exactly two node vertices, v_o must lie either between u_0 and v, or between v and u_1. So, either

$$\gamma_1(v_o) = u_0 \text{ and } \gamma_2(v_o) = v,$$

or

$$\gamma_1(v_o) = u_1 \text{ and } \gamma_2(v_o) = v.$$

In the first case (see fig. 10.6), u_1 replaces v as the node city which is connected to v_o:

$$\gamma_2(v_o) \leftarrow u_1.$$

In the second case u_0 replaces v as the node city which is connected to v_o:

$$\gamma_2(v_o) \leftarrow u_0.$$

Furthermore we need to add *START* and *END* values at v_o for the new (compound) edges we created earlier. In the first case for the edges e_{01} and e_{10} that we created replacing the edges e_i and e_{j+1} between u_0 and v. In the second case for the edges e_{01} and e_{10} that we created replacing the edges e_{i+1} and e_j between v and u_1.

In the first case, the *END* value of the edge e_{01} at v_o becomes the *END* value of e_i at v_o, and the *START* value of e_{01} at v_o becomes the *START* value of e_i at v_o. Furthermore the *END* value of the edge e_{10} at v_o becomes the *END* value of e_{j+1} at v_o, and the *START* value of e_{10} at v_o becomes the *START* value of e_{j+1} at v_o:

$$\begin{aligned}
&END(u_0, v_o, e_{01}) \leftarrow END(u_0, v_o, e_i),\\
&START(v_o, u_1, e_{01}) \leftarrow START(v_o, v, e_i),\\
&END(u_1, v_o, e_{10}) \leftarrow END(v, v_o, e_{j+1}),\\
&START(v_o, u_0, e_{10}) \leftarrow START(v_o, u_0, e_{j+1}).
\end{aligned}$$

In the second case, the *END* value of the edge e_{10} at v_o becomes the *END* value of e_j at v_o, and the *START* value of e_{10} at v_o becomes the *START* value of e_j at v_o. Furthermore the *END* value of the edge e_{01} at v_o becomes the *END* value of e_{i+1} at v_o, and the *START* value of e_{01} at v_o becomes the *START* value of e_{i+1} at v_o:

$$\begin{aligned}
&END(u_1, v_o, e_{10}) \leftarrow END(u_1, v_o, e_j),\\
&START(v_o, u_0, e_{10}) \leftarrow START(v_o, v, e_j),\\
&END(u_0, v_o, e_{01}) \leftarrow END(v, v_o, e_{i+1}),\\
&START(v_o, u_1, e_{01}) \leftarrow START(v_o, u_1, e_{i+1}).
\end{aligned}$$

10.4.2. The algorithm to transform a discrete dynamic network

We now give a formal definition of the algorithm to transform a discrete dynamic network into a discrete dynamic network with offset cities, which can be equivalently but more efficiently searched for an optimal path.

Consider a finite discrete dynamic network consisting of the graph (V, E) and the connection function *CON*. We shall construct a discrete dynamic network with offset vertices, consisting of V_{node}, V_{off}, E', *CON*, *START*, *END*. V_{node} is the collection of node vertices, V_{off} the collection of offset vertices, E' the collection of directed edges joining two vertices from V_{node}, *CON* the connection function. The *START* and *END*

functions give the *start* and *end* values of the edges connected to offset vertices. With each offset vertex $v \in V_{off}$, we associate 2 values, $\gamma_1(v)$ and $\gamma_2(v)$, which denote the two different node vertices which are connected to the offset vertex.

(1) $V_{off} \leftarrow \emptyset$, $V_{node} \leftarrow \emptyset$.
(2) $E' \leftarrow E$. Remove from E' all self-loops.
(3) for each vertex $v \in V$,
    if there exist two vertices u_0 and u_1, $u_0 \neq v$, $u_1 \neq v$, $u_0 \neq u_1$,
        such that
            for all edges e_k: $u_p \rightarrow v$ we have $u_p = u_0$ or $u_p = u_1$, and
            for all edges e_l: $v \rightarrow u_q$ we have $u_q = u_0$ or $u_q = u_1$, and
    if each edge e_i: $u_0 \rightarrow v$, has a connecting edge e_{i+1}: $v \rightarrow u_1$, with $u_0 \neq u_1$,
    and each edge e_{i+1}: $v \rightarrow u_1$, has a preceding edge e_i: $u_0 \rightarrow v$,
        such that
            $start(e_{i+1}) \geq end(e_i) + CON(v, e_i, e_{i+1})$,
    and if each edge e_j: $u_1 \rightarrow v$, has a connecting edge e_{j+1}: $v \rightarrow u_0$,
    and each edge e_{j+1}: $v \rightarrow u_0$, has a preceding edge e_j: $u_1 \rightarrow v$,
        such that
            $start(e_{j+1}) \geq end(e_j) + CON(v, e_j, e_{j+1})$,
    then $\gamma_1(v) \leftarrow u_0$ and $\gamma_2(v) \leftarrow u_1$.
        $V_{off} \leftarrow V_{off} + \{ v \}$.
        $V_{node} \leftarrow V_{node} + \{ u_0, u_1 \}$.
        for each pair of edges e_i and e_{i+1} as described above:
            $E' \leftarrow E' - \{ e_i, e_{i+1} \}$.
            create an edge e_{01}: $u_0 \rightarrow u_1$,
                $start(e_{01}) \leftarrow start(e_i)$,
                $end(e_{01}) \leftarrow end(e_{i+1})$,
                $E' \leftarrow E' + \{ e_{01} \}$,
                $END(u_0, v, e_{01}) \leftarrow end(e_i)$,
                $START(v, u_1, e_{01}) \leftarrow start(e_{i+1})$.
                for each edge e for which $CON(u_0, e, e_i)$ is defined:
                    $CON(u_0, e, e_{01}) \leftarrow CON(u_0, e, e_i)$.
                for each edge e for which $CON(u_1, e_{i+1}, e)$ is defined:
                    $CON(u_1, e_{01}, e) \leftarrow CON(u_1, e_{i+1}, e)$.
        for each pair of edges e_j and e_{j+1} as described above:
            $E' \leftarrow E' - \{ e_j, e_{j+1} \}$.
            create an edge e_{10}: $u_1 \rightarrow u_0$,
                $start(e_{10}) \leftarrow start(e_j)$,

$end(e_{10}) \leftarrow end(e_{j+1})$,
$E' \leftarrow E' + \{ e_{10} \}$,
$END(u_1, v, e_{10}) \leftarrow end(e_j)$,
$START(v, u_0, e_{10}) \leftarrow start(e_{j+1})$.
for each edge e for which $CON(u_1, e, e_j)$ is defined:
$CON(u_1, e, e_{10}) \leftarrow CON(u_1, e, e_j)$.
for each edge e for which $CON(u_0, e_{j+1}, e)$ is defined:
$CON(u_0, e_{10}, e) \leftarrow CON(u_0, e_{j+1}, e)$.

if $v \in V_{node}$ then there exist offset vertices v_o for which v was a node vertex.
for each such offset vertex v_o:
if $\gamma_1(v_o) = u_0$ and $\gamma_2(v_o) = v$.
then $\gamma_2(v_o) \leftarrow u_1$,
for each edge e_i replaced above, for which $END(u_0, v_o, e_i)$ is defined:
$END(u_0, v_o, e_{01}) \leftarrow END(u_0, v_o, e_i)$,
$START(v_o, u_1, e_{01}) \leftarrow START(v_o, v, e_i)$.
for each edge e_{j+1} replaced above, for which $END(v, v_o, e_{j+1})$ is defined:
$END(u_1, v_o, e_{10}) \leftarrow END(v, v_o, e_{j+1})$,
$START(v_o, u_0, e_{10}) \leftarrow START(v_o, u_0, e_{j+1})$.
else $\gamma_2(v_o) \leftarrow u_0$,
for each edge e_j replaced above, for which $END(u_1, v_o, e_j)$ is defined:
$END(u_1, v_o, e_{10}) \leftarrow END(u_1, v_o, e_j)$,
$START(v_o, u_0, e_{10}) \leftarrow START(v_o, v, e_j)$.
for each edge e_{i+1} replaced above, for which $END(v, v_o, e_{i+1})$ is defined:
$END(u_0, v_o, e_{01}) \leftarrow END(v, v_o, e_{i+1})$,
$START(v_o, u_1, e_{01}) \leftarrow START(v_o, u_1, e_{i+1})$.
$V_{node} \leftarrow V_{node} - \{ v \}$.
else $V_{node} \leftarrow V_{node} + \{ v \}$.

10.4.3. Searching a discrete dynamic network with offset vertices

We now adapt the DYNET algorithm for searching a discrete dynamic network from chapter 6 to allow an offset vertex as starting vertex or terminating vertex. As we saw earlier, we only need to consider offset vertices during search if the starting vertex or terminating vertex is an offset vertex. We shall look at each case in turn, for the forward pass and for the backward pass.

10.4.3.1. An offset vertex as the starting vertex; the forward pass

In the forward pass of the definition of the DYNET algorithm (see section 6.6.), the starting vertex is handled in step (1): a partial path is created, consisting of the starting vertex only. The *end* value of the path is made the start value T_{start}. Since this path is the only path that is put in the frontier, at step (5) new (partial) paths to all neighbours of the starting vertex are created. If the starting vertex is an offset vertex, we create paths from the offset vertex to the two node vertices adjacent to the offset vertex. The end values of the paths are the *end* values of the edges from the offset vertex to the node vertices which are connected to it. The offset vertex is temporarily treated as a node vertex by creating paths to its adjacent node vertices. The information of the edges from the offset vertex to the node vertex can be recovered by using the *START* and *END* functions to determine the *start* and *end* values of the connecting edges. Of course we have to make sure that we include all *relevant* edges: all edges which arrive within the *maxiCON* interval at the node vertex. Precisely:

For each edge e for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_1(s), e) \geq T_{start}$,
(2) $end(e) < end(e_{min}) + maxiCON(\gamma_1(s))$,
where $end(e_{min})$ is the minimum *end* value of any edge satisfying (1).
create a partial path $P = s, e, \gamma_1(s)$.

The same must also be repeated for the second node vertex to which the starting vertex is connected, $\gamma_2(s)$:

For each edge e for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_2(s), e) \geq T_{start}$,
(2) $end(e) < end(e_{min}) + maxiCON(\gamma_2(s))$,
where $end(e_{min})$ is the minimum *end* value of any edge satisfying (1).
create a partial path $P = s, e, \gamma_2(s)$.

10.4.3.2. An offset vertex as the terminating vertex; the forward pass

In the forward pass, when the terminating vertex is an offset vertex, we create partial paths to the offset vertex when a partial path to one of its adjacent node vertices has become the branching path (step (3) of the algorithm of section 6.6). We can recognize a node vertex adjacent to the offset vertex by comparing it to the γ values of the offset vertex. We can recover the information about the edges connecting the node vertex and the offset vertex by using the *START* and *END* functions to determine the *start* and *end* values of the connecting edges. We only

need the edge which arrives at the offset vertex with the smallest *end* value. Since, from the offset vertex, we shall not construct further paths, we do **not** need to include other *relevant* edges (edges which arrive within the *maxiCON* interval at the offset vertex). Precisely:

if $u_j = \gamma_1(t)$ or if $u_j = \gamma_2(t)$,
then find the edge e_j: $u_j \rightarrow \gamma_2(t)$ or $u_j \rightarrow \gamma_1(t)$ respectively, for which the following three *ordered* conditions hold:
(1) $END(u_j, t, e_j)$ is defined,
(2) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$,
(3) $END(u_j, t, e_j)$ is minimum.

10.4.3.3. Offset vertices as starting and terminating vertex; the forward pass

When both the starting vertex and the terminating vertex are an offset vertex, a special case arises when they are both adjacent to the same pair of node vertices. We must (also) create a partial path from one offset vertex to the other directly, without going to a node vertex first. We must find an edge which has a *start* value at the (offset) starting vertex of at least T_{start}, and which has a minimum *end* value at the (offset) terminating vertex. These values of the edge at the offset vertices can be recovered by using the *START* and *END* functions. In order to determine in which direction we must go from one node vertex to the other node vertex, we can also use the *START* and *END* values. We shall look at each possible case (see also fig. 10.8).

(1) if $\gamma_1(s) = \gamma_1(t)$ and $\gamma_2(s) = \gamma_2(t)$, and if there exists an edge e such that
$START(s, \gamma_2(s), e) < END(\gamma_1(t), t, e)$,
then we must find the edge e from $\gamma_1(s)$ to $\gamma_2(t)$ for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_2(s), e) \geq T_{start}$
(2) $END(\gamma_1(t), t, e)$ is minimum.

(2) if $\gamma_1(s) = \gamma_1(t)$ and $\gamma_2(s) = \gamma_2(t)$, and if there exists an edge e such that
$START(s, \gamma_2(s), e) \geq END(\gamma_1(t), t, e)$,
then we must find the edge e from $\gamma_2(s)$ to $\gamma_1(t)$ for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_1(s), e) \geq T_{start}$
(2) $END(\gamma_2(t), t, e)$ is minimum.

(3) if $\gamma_1(s) = \gamma_2(t)$ and $\gamma_2(s) = \gamma_1(t)$, and if there exists an edge e such that

$START(s, \gamma_2(s), e) < END(\gamma_2(t), t, e)$,

then we must find the edge e from $\gamma_1(s)$ to $\gamma_1(t)$ for which the following two *ordered* conditions hold:

(1) $START(s, \gamma_2(s), e) \geq T_{start}$

(2) $END(\gamma_2(t), t, e)$ is minimum.

(4) if $\gamma_1(s) = \gamma_2(t)$ and $\gamma_2(s) = \gamma_1(t)$, and if there exists an edge e such that

$START(s, \gamma_2(s), e) \geq END(\gamma_2(t), t, e)$,

then we must find the edge e from $\gamma_2(s)$ to $\gamma_2(t)$ for which the following two *ordered* conditions hold:

(1) $START(s, \gamma_1(s), e) \geq T_{start}$

(2) $END(\gamma_1(t), t, e)$ is minimum.

10.4.3.4. An offset vertex as the starting vertex; the backward pass

In the backward pass, when the starting vertex is an offset vertex, we create partial paths from the offset vertex when a partial path to one of its adjacent node vertices has become the branching path (step (3) of the algorithm of section 6.6). We only need the edge which arrives at the offset vertex with the greatest *start* value. Since we shall not construct further paths to the offset vertex, we do **not** need to include other *relevant* edges (edges which depart within the *maxiCON* interval at the offset vertex). Precisely:

if $u_j = \gamma_1(s)$ (or respectively if $u = \gamma_2(s)$):

then find the edges e_{j-1}: $\gamma_2(s) \rightarrow u_j$ (respectively $\gamma_1(s) \rightarrow u_j$), for which the following three *ordered* conditions hold:

(1) $START(s, u_j, e_{j-1})$ is defined,

(2) $end(e_{j-1}) \leq start(e_j) - CON(u_j, e_{j-1}, e_j)$,

(3) $START(s, u_j, e_{j-1})$ is maximum.

10.4.3.5. An offset vertex as the terminating vertex; the backward pass

In the backward pass of the definition of the DYNET algorithm, the terminating vertex is handled in step (1): a partial path is created, consisting of the starting vertex only. The *start* value of the path is made the label of the terminating vertex from the forward pass $\lambda(t)$. Since this path is the only path that is put in the frontier, at step (5) new (partial) paths from all neighbours of the terminating vertex are created. If the terminating vertex is an offset vertex, we create paths to the offset vertex from the node vertices adjacent to the offset vertex. The start values of the paths are the *start* values of the edges from the adjacent node vertices to the offset vertex. The offset

vertex is temporarily treated as a node vertex by creating paths from its adjacent node vertices. The information of the edges from the node vertex to the offset vertex can be recovered by using the *START* and *END* functions to determine the *start* and *end* values of the connecting edges. Of course we have to make sure that we include all *relevant* edges: all edges which depart within the *maxiCON* interval at the node vertex. Precisely:

For each edge e for which the following two *ordered* conditions hold:
(1) $END(\gamma_1(t), t, e) \leq \lambda(t)$,
(2) $start(e) < start(e_{max}) - maxiCON(\gamma_1(t))$,
where $end(e_{max})$ is the maximum *start* value of any edge satisfying (1),
create a partial path $P = \gamma_1(t), e, t$.

10.4.3.6. Offset vertices as starting and terminating vertex; the backward pass

In the backward pass, when both the starting vertex and the terminating vertex are offset vertices, and both adjacent to the same pair of node vertices, again we must (also) create a partial path from one offset vertex to the other directly, without going to a node vertex first. We must find an edge which has an *end* value at the (offset) terminating vertex of at most $\lambda(t)$, and which has a maximum *start* value at the (offset) starting vertex. These values of the edge at the offset vertices can be recovered by using the *START* and *END* functions. In order to determine in which direction we must go from one node vertex to the other node vertex, we can also use the *START* and *END* values. We shall look at each possible case (see also fig. 10.8).

(1) if $\gamma_1(s) = \gamma_1(t)$ and $\gamma_2(s) = \gamma_2(t)$, and if there exists an edge e such that
$START(s, \gamma_2(s), e) < END(\gamma_1(t), t, e)$,
then we must find the edge e from $\gamma_1(s)$ to $\gamma_2(t)$ for which the following two *ordered* conditions hold:
(1) $END(\gamma_1(t), t, e) \leq \lambda(t)$
(2) $START(s, \gamma_2(s), e)$ is maximum.

(2) if $\gamma_1(s) = \gamma_1(t)$ and $\gamma_2(s) = \gamma_2(t)$, and if there exists an edge e such that
$START(s, \gamma_2(s), e) \geq END(\gamma_1(t), t, e)$,
then we must find the edge e from $\gamma_2(s)$ to $\gamma_1(t)$ for which the following two *ordered* conditions hold:
(1) $END(\gamma_2(t), t, e) \leq \lambda(t)$
(2) $START(s, \gamma_1(s), e)$ is maximum.

(3) if $\gamma_1(s) = \gamma_2(t)$ and $\gamma_2(s) = \gamma_1(t)$, and if there exists an edge e such that

$START(s, \gamma_2(s), e) < END(\gamma_2(t), t, e)$,

then we must find the edge e from $\gamma_1(s)$ to $\gamma_1(t)$ for which the following two *ordered* conditions hold:

(1) $END(\gamma_2(t), t, e) \leq \lambda(t)$

(2) $START(s, \gamma_2(s), e)$ is maximum.

(4) if $\gamma_1(s) = \gamma_2(t)$ and $\gamma_2(s) = \gamma_1(t)$, and if there exists an edge e such that

$START(s, \gamma_2(s), e) \geq END(\gamma_2(t), t, e)$,

then we must find the edge e from $\gamma_2(s)$ to $\gamma_2(t)$ for which the following two *ordered* conditions hold:

(1) $END(\gamma_1(t), t, e) \leq \lambda(t)$

(2) $START(s, \gamma_1(s), e)$ is maximum.

10.4.4. DYNET for searching a discrete dynamic network with offset vertices

We now give a formal definition of the DYNET algorithm, adapted to handle an offset vertex as starting vertex or terminating vertex. An offset vertex as starting vertex is handled in step (1) of the forward pass and step (5) of the backward pass. An offset vertex as terminating vertex is handled in step (5) of the forward pass and in step (1) of the backward pass. The special case of both the starting vertex and the terminating vertex as an offset vertex between the same pair of node vertices is handled in step (2) of both passes. The other steps are similar to the original definition of the algorithm for searching a discrete dynamic network (for an explanation of the different steps see chapter 6).

Consider a discrete dynamic network containing offset vertices, consisting of V_{node}, V_{off}, E, CON, $START$, END. V_{node} is the collection of node vertices, V_{off} the collection of offset vertices, E the collection of directed edges joining two vertices from V_{node}, CON the connection function. The two special vertices s and t of the network are the *starting* and *terminating* vertices (both either node or offset vertices). We want to find a legal path from s to t in our discrete dynamic network, where the end value of the path is minimum and given this end value, the start value of the path is maximum and at least T_{start}. The maximum value of CON at a vertex u is *maxiCON(u)*, which is non-zero. The $START$ and END functions give the *start* and *end* values of the edges connected to offset vertices.

Pass 1:

(1) for all $v \in V_{\text{node}}$, $\lambda(v) \leftarrow \infty$, and $\omega(v, u) = \infty$ for each neighbour u of v.

if $t \in V_{\text{off}}$ then $\lambda(t) \leftarrow \infty$.

if $s \in V_{\text{node}}$

then $\lambda(s) \leftarrow T_{start}$ and create a partial path P_0 consisting of s only,
$end(P_0) \leftarrow T_{start}$. $F \leftarrow \{ P_0 \}$.
else
$F \leftarrow \varnothing$.
for each edge e for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_1(s), e) \geq T_{start}$,
(2) $end(e) < end(e_{min}) + maxiCON(\gamma_1(s))$,
where $end(e_{min})$ is the minimum *end* value of any edge satisfying (1).
create a partial path $P = s, e, \gamma_1(s)$, $F \leftarrow F + \{ P \}$,
if $\lambda(\gamma_1(s)) > end(e)$ then $\lambda(\gamma_1(s)) \leftarrow end(e)$.
for each edge e for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_2(s), e) \geq T_{start}$,
(2) $end(e) < end(e_{min}) + maxiCON(\gamma_2(s))$,
where $end(e_{min})$ is the minimum *end* value of any edge satisfying (1).
create a partial path $P = s, e, \gamma_2(s)$, $F \leftarrow F + \{ P \}$,
if $\lambda(\gamma_2(s)) > end(e)$ then $\lambda(\gamma_2(s)) \leftarrow end(e)$.
(2) if $s \in V_{off}$ and $t \in V_{off}$:
if $\gamma_1(s) = \gamma_1(t)$ and $\gamma_2(s) = \gamma_2(t)$, then
if there exists an edge e such that $START(s, \gamma_2(s), e) < END(\gamma_1(t), t, e)$,
then find the edge for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_2(s), e) \geq T_{start}$
(2) $END(\gamma_1(t), t, e)$ is minimum.
and create a partial path $P = s, e, t$, $end(P) \leftarrow END(\gamma_1(t), t, e)$.
else find the edge for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_1(s), e) \geq T_{start}$
(2) $END(\gamma_2(t), t, e)$ is minimum,
and create a partial path $P = s, e, t$, $end(P) \leftarrow END(\gamma_2(t), t, e)$.
else
if there exists an edge e such that $START(s, \gamma_2(s), e) < END(\gamma_2(t), t, e)$,
then find the edge for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_2(s), e) \geq T_{start}$
(2) $END(\gamma_2(t), t, e)$ is minimum,
and create a partial path $P = s, e, t$, $end(P) \leftarrow END(\gamma_2(t), t, e)$.
else find the edge for which the following two *ordered* conditions hold:
(1) $START(s, \gamma_1(s), e) \geq T_{start}$
(2) $END(\gamma_1(t), t, e)$ is minimum,
and create a partial path $P = s, e, t$, $end(P) \leftarrow END(\gamma_1(t), t, e)$.
$F \leftarrow F + \{ P \}$. If $\lambda(t) > end(P)$ then $\lambda(t) \leftarrow end(P)$.

(3) let P_m be a partial path $s, e_0, u_1, ..., u_{j-1}, e_{j-1}, u_j$ in F for which $end(P_m)$ is minimum; if F is empty then stop, no complete path could be found.
(4) if $u_j = t$, stop, P_m is a complete path with an optimal end value.
(5) if $t \in V_{off}$, then
if $u_j = \gamma_1(t)$ or if $u = \gamma_2(t)$:
then find the edge e_j: $u_j \rightarrow \gamma_2(t)$ or $u_j \rightarrow \gamma_1(t)$ respectively, for which the following three *ordered* conditions hold:
(1) $END(u_j, t, e_j)$ is defined,
(2) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$,
(3) $END(u_j, t, e_j)$ is minimum.
create a partial path $P_n = s, e_0, u_1, ..., u_{j-1}, e_{j-1}, u_j, e_j, t$,
$end(P_n) \leftarrow END(u_j, t, e_j)$, and $F \leftarrow F + \{ P_n \}$.
if $\lambda(t) > END(u_j, t, e_j)$ then $\lambda(t) \leftarrow END(u_j, t, e_j)$.
(6) if $end(P_m) < \lambda(u_j) + maxiCON(u_j)$.
then for each relevant edge e_j : $u_j \rightarrow u_{j+1}$:
if $\lambda(u_{j+1}) > end(e_j)$
then $\lambda(u_{j+1}) \leftarrow end(e_j)$
if $end(e_j) < \lambda(u_{j+1}) + maxiCON(u_{j+1})$
then create a partial path $P_n = s, e_0, u_1, ..., u_{j-1}, e_{j-1}, u_j, e_j, u_{j+1}$, and
$F \leftarrow F + \{ P_n \}$.
if $end(e_j) < \omega(u_{j+1}, u_j)$
then $\omega(u_{j+1}, u_j) \leftarrow end(e_j)$.
(7) $F \leftarrow F - \{ P_m \}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_0, ..., u_{j-1}, e_{j-1}, u_j$ and a vertex u_{j+1}, then the relevant edges from u_j to u_{j+1} are the edges e_j: $u_j \rightarrow u_{j+1}$ for which the following two *ordered* conditions hold:
(1) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$,
(2) $end(e_j) < end(e_{min}) + maxiCON(u_{j+1})$,
where $end(e_{min})$ is the minimum end value of any edge satisfying (1).

Pass 2:

(1) for all $v \in V_{node}$, $\kappa(v) \leftarrow -\infty$.
if $s \in V_{off}$ then $\kappa(s) \leftarrow -\infty$.
if $t \in V_{node}$
then $\kappa(t) \leftarrow \lambda(t)$ and create a partial path P_0 consisting of t only,
$start(P_0) \leftarrow \lambda(t)$. $F \leftarrow \{ P_0 \}$.
else
$F \leftarrow \varnothing$,

for each edge e for which the following two *ordered* conditions hold:
(1) $END(\gamma_1(t), t, e) \le \lambda(t)$,
(2) $start(e) < start(e_{max}) - maxiCON(\gamma_1(t))$,
where $end(e_{max})$ is the maximum *start* value of any edge satisfying (1).
create a partial path $P = \gamma_1(t), e, t$, $F \leftarrow F + \{ P \}$,
if $\kappa(\gamma_1(t)) < start(e)$ then $\kappa(\gamma_1(t)) \leftarrow start(e)$.
for each edge e for which the following two *ordered* conditions hold:
(1) $END(\gamma_2(t), t, e) \le \lambda(t)$,
(2) $start(e) < start(e_{max}) - maxiCON(\gamma_2(t))$,
where $end(e_{max})$ is the maximum *start* value of any edge satisfying (1).
create a partial path $P = \gamma_2(t), e, t$, $F \leftarrow F + \{ P \}$,
if $\kappa(\gamma_2(t)) < start(e)$ then $\kappa(\gamma_2(t)) \leftarrow start(e)$.
(2) if $s \in V_{off}$ and $t \in V_{off}$:
if $\gamma_1(s) = \gamma_1(t)$ and $\gamma_2(s) = \gamma_2(t)$, then
if there exists an edge e such that $START(s, \gamma_2(s), e) < END(\gamma_1(t), t, e)$,
then find the edge for which the following two *ordered* conditions hold:
(1) $END(\gamma_1(t), t, e) \le \lambda(t)$
(2) $START(s, \gamma_2(s), e)$ is maximum,
and create a partial path $P = s, e, t$, $start(P) \leftarrow START(s, \gamma_2(s), e)$.
else find the edge for which the following two *ordered* conditions hold:
(1) $END(\gamma_2(t), t, e) \le \lambda(t)$
(2) $START(s, \gamma_1(s), e)$ is maximum,
and create a partial path $P = s, e, t$, $start(P) \leftarrow START(s, \gamma_1(s), e)$.
else
if there exists an edge e such that $START(s, \gamma_2(s), e) < END(\gamma_2(t), t, e)$,
then find the edge for which the following two *ordered* conditions hold:
(1) $END(\gamma_2(t), t, e) \le \lambda(t)$
(2) $START(s, \gamma_2(s), e)$ is maximum,
and create a partial path $P = s, e, t$, $start(P) \leftarrow START(s, \gamma_2(s), e)$.
else find the edge for which the following two *ordered* conditions hold:
(1) $END(\gamma_1(t), t, e) \le \lambda(t)$
(2) $START(s, \gamma_1(s), e)$ is maximum,
and create a partial path $P = s, e, t$, $start(P) \leftarrow START(s, \gamma_1(s), e)$.
$F \leftarrow F + \{ P \}$. If $\lambda(t) > end(P)$ then $\lambda(t) \leftarrow end(P)$.
(3) let P_m be a partial path $u_j, \ldots, u_{k-1}, e_{k-1}, t$ in F for which $start(P_m)$ is maximum.
(4) if $u_j = s$, stop, P_m is an optimal complete path.
(5) if $s \in V_{off}$, then
if $u_j = \gamma_1(s)$ or if $u = \gamma_2(s)$:

then find the edge e_{j-1}: $\gamma_2(s) \rightarrow u_j$ or $\gamma_1(s) \rightarrow u_j$ respectively, for which the following three *ordered* conditions hold:

(1) $START(s, u_j, e_{j-1})$ is defined,

(2) $end(e_{j-1}) \leq start(e_j) - CON(u_j, e_{j-1}, e_j)$,

(3) $START(s, u_j, e_{j-1})$ is maximum.

create a partial path $P_n = s, e_{j-1}, u_j, \ldots, u_{k-1}, e_{k-1}, t$,

$start(P_n) \leftarrow START(s, u_j, e_{j-1})$, and $F \leftarrow F + \{ P_n \}$.

if $\kappa(t) < START(s, u_j, e_{j-1})$ then $\kappa(t) \leftarrow START(s, u_j, e_{j-1})$.

(6) if $start(P_m) > \kappa(u_j) - maxiCON(u_j)$

then for each relevant edge e_{j-1} : $u_{j-1} \rightarrow u_j$, with $\omega(u_j, u_{j-1}) \leq start(P_m)$,

if $\kappa(u_{j-1}) < start(e_{j-1})$

then $\kappa(u_{j-1}) \leftarrow start(e_{j-1})$

if $start(e_{j-1}) > \kappa(u_{j-1}) - maxiCON(u_{j-1})$

then create a partial path $P_n = u_{j-1}, e_{j-1}, u_j, \ldots, u_{k-1}, e_{k-1}, t$, and

$F \leftarrow F + \{ P_n \}$.

(7) $F \leftarrow F - \{ P_m \}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_j, e_j, u_{j+1}, \ldots, u_k$ and a vertex u_{j-1}, then the relevant edges from u_{j-1} to u_j are the edges e_{j-1}: $u_{j-1} \rightarrow u_j$ for which the following two *ordered* conditions hold:

(1) $end(e_{j-1}) \leq start(e_j) - CON(u_j, e_{j-1}, e_j)$,

(2) $start(e_{j-1}) > start(e_{max}) - maxiCON(u_{j-1})$,

where $start(e_{max})$ is the maximum start value of any edge satisfying (1).

11. Train Changes

Until now we have not taken into account one particular property of travelling by train: (explicit) train changes. We have been interested in optimal solutions in terms of travel time only. In a practical application, however, people are also interested in the number of train changes. An optimal solution in terms of travel time should also have the least number of train changes possible, given the optimal travel time. Furthermore, people are usually also interested in solutions which may take more travel time, but which have fewer train changes. In this chapter, we shall first look at how we can minimize the number of train changes in an optimal solution. Then we shall look at how we can find some suboptimal solutions with fewer train changes.

11.1. A train in a discrete dynamic network

In the previous chapters, conceptually we changed train at each station; every time we used an edge it was regarded as a next train. When no train change was actually required this was implicitly represented by the connection function: if e_i and e_{i+1} represent the same ongoing train at the station represented by the vertex v_{i+1}, then $CON(v_{i+1}, e_i, e_{i+1}) = 0$.

Note that even though we may need no time between arrival and departure in order to continue on the same train, the train may be standing at the station for some time. Precisely, even though

$$CON(v_{i+1}, e_i, e_{i+1}) = 0$$

it may be so that $start(e_{i+1}) - end(e_i) > 0$.
It might even be so that $start(e_{i+1}) - end(e_i) > maxiCON(v_{i+1})$.

In order to model a train change more accurately, we first have to introduce the concept of a train to discrete dynamic networks.

In order to represent a train in a discrete dynamic network we give each edge e an extra attribute, the identifier $id(e)$. A train is a sequence of edges $e_0, e_1, \ldots, e_k$ such that:

(1) the start vertex of e_{i+1} is the end vertex of e_i , $0 \leq i < k$,
(2) $start(e_{i+1}) \geq end(e_i)$, $0 \leq i < k$,
(3) $CON(v_{i+1}, e_i, e_{i+1}) = 0$ where v_{i+1} is the end vertex of e_i and the start vertex of e_{i+1} , $0 \leq i < k$,

Each edge e_i in the sequence, $0 \leq i \leq k$, is given the same identifier $id(e_i)$.

A train is a sequence of connecting edges which require no connection time. The edges which compose the train have the same identifier. This way, a train can be viewed as a macro operator: one super edge consisting of multiple connecting edges (for a discussion of macro operators see [Da, 1977]).

11.2. A train change in a discrete dynamic network

Suppose we have the following (legal) path P in a discrete dynamic network:

$$v_0, e_0, v_1, \ldots, v_j, e_j, v_{j+1}, \ldots, e_{k-1}, v_k$$

A connection v_{i+1}, e_i, e_{i+1} , $0 \leq i < k$, is a train change if

$$id(e_{i+1}) \neq id(e_i)$$

or equivalently,

$$CON(v_{i+1}, e_i, e_{i+1}) > 0.$$

11.3. Solutions with unnecessary train changes

Sometimes, unnecessary train changes occur in a solution found by using the DYNET algorithm for searching discrete dynamic network (presented in chapter 6). For instance, consider the following example, in which we want to travel from Utg to Asd, departing at 7:00. The changing time at Zd and Ass is 5 minutes.

	100	105	110	115	Station
Utg	7:00	7:10			Uitgeest
Zd	7:15	7:25	7:30		Zaandam
Ass		7:35	7:40	7:45	Amsterdam Sloterdijk
Asd				7:50	Amsterdam Central Station

The forward pass of the algorithm, which determines the earliest possible arrival, gives as solution to travel from Utg to Zd by train 100, from Zd to Ass by

train 105 and from Ass to Asd by train 115. The backward pass of the algorithm, which determines the matching latest possible departure, gives as solution to travel from Utg to Zd by train 105, from Zd to Ass by train 110 and from Ass to Asd by train 115. This (final) solution has two train changes, whereas we could travel equally optimally from Utg to Ass by train 105 and from Ass to Asd by train 115 with only one train change at Ass. Such an unnecessary train change is caused by the "greediness" of the algorithm. When we arrive at a station, we take the very first opportunity to travel onwards (the relevant edges). We cannot tell beforehand which non-relevant later edges (trains) might give an equally optimal solution with fewer train changes. Trying all later non-relevant edges during the search process could result in a combinatorial explosion. Therefore, in order to eliminate unnecessary train changes, we shall postprocess a solution.

11.4. Eliminating unnecessary train changes

Suppose that, using the algorithm for searching a discrete dynamic network, we have found the following (legal) path P as solution:

$$s = v_0, e_0, v_1, \ldots, v_j, e_j, v_{j+1}, \ldots, e_{k-1}, v_k = t$$

We must process this path P in such a way that:

(1) there is a minimum number of train changes,
(2) the path P remains optimal,
(3) the path P remains legal.

We shall first look at each of these aspects separately, and then combine them into one algorithm.

11.4.1. Eliminating a train change

In order to eliminate unnecessary train changes we traverse the path P in a forward fashion (since the final solution was found by a backward search process). At each vertex v_{i+1}, $0 \leq i < k-1$, we do the following check:

if $id(e_{i+1}) \neq id(e_i)$
then
  if there exists an edge e_q: $v_{i+1} \rightarrow v_{i+2}$ such that $id(e_q) = id(e_i)$,
  then replace e_{i+1} by e_q.

At each vertex we check whether there is a train change. If there is, then we check whether the previous train continues to the same station as the next train, in which case the journey can actually be continued on the previous train.

11.4.2. Preserving optimality of solution

In the process we must, however, take care to preserve optimality of solution. For instance, consider the following example, in which we want to travel from Hk to Asd. The changing time at Hlm is 4 minutes.

	105	200	Station
Hk		8:00	Heemskerk
Hlm		8:16	Haarlem (arrival)
Hlm	8:20	8:23	Haarlem (departure)
Asd	8:35	8:38	Amsterdam Central Station

In this example, the search algorithm would give as solution to travel from Hk to Hlm by train 200, and from Hlm to Asd by train 105 (arriving at 8:35). If we would traverse this solution as described in the previous section, we would replace train 105 from Hlm to Asd by train 200 (resulting in an arrival at 8:38). Although this is a solution with one fewer train change, it has become suboptimal. So, when the edge we try to replace by e_q arrives at the terminating vertex v_k, we must also test whether $end(e_q) \leq end(e_{k-1})$. In a case like this, when a train stands at a station longer than the time required to change to another train (possibly also going to our destination), the solution may become suboptimal. Another case in which this might occur is when the ongoing train to our destination is slower than the train we changed to.

11.4.3. Preserving legality of solution

We should also take care to preserve legality. For instance, consider the following example, in which we want to travel from Hk to Ut. The changing time at Hlm is 4 minutes and at Asd 5 minutes.

	400	500	Station
Hk	9:00		Heemskerk
Hlm	9:20	9:25	Haarlem
Asd	9:45	9:47	Amsterdam Central Station
Ut		10:12	Utrecht Central Station

In this example, the search algorithm would give as solution to travel from Hk to Hlm by train 400, and from Hlm to Ut by train 500. If we would traverse this solution as described above, we would replace train 500 from Hlm to Asd by train

400. However, that would yield an illegal path, since the margin at Asd is only 2 minutes whereas the changing time is 5 minutes. We cannot simply test whether the next change would have a sufficient margin (test whether $start(e_{i+2}) - end(e_q) \geq CON(v_{i+2}, e_q, e_{i+2})$), because it may be possible to also replace the next train e_{i+2}, in which case we would not have to change trains at all. Instead, we must keep track of which illegal changes may occur, and backtrack when we cannot replace a next train resulting in the illegal change. We can describe a backtracking process by using a recursive definition.

11.4.4. The algorithm to eliminate unnecessary train changes

We now combine the techniques discussed in the previous sections into one recursively defined process *TRAVERSE(P, i)*:

Suppose $P = v_0, e_0, v_1, \ldots, v_j, e_j, v_{j+1}, \ldots, e_{k-1}, v_k$.
if $i = k-1$ then $TRAVERSE(P, i) \leftarrow TRUE$.
else
  if $id(e_{i+1}) \neq id(e_i)$
  then
    if there exists an edge e_q: $v_{i+1} \rightarrow v_{i+2}$ such that
      (1) $id(e_q) = id(e_i)$ and
      (2) $end(e_q) \leq end(e_{i+1})$ if $i+2 = k$.
    then
      replace e_{i+1} by e_q,
      if $TRAVERSE(P, i+1) = FALSE$
        if $start(e_{i+2}) - end(e_q) < CON(v_{i+2}, e_q, e_{i+2})$
        then
          replace back e_q by e_{i+1},
          $TRAVERSE(P, i+1)$,
          $TRAVERSE(P, i) \leftarrow FALSE$.
        else $TRAVERSE(P, i) \leftarrow TRUE$.
      else $TRAVERSE(P, i) \leftarrow TRUE$.
    else $TRAVERSE(P, i+1)$,
      $TRAVERSE(P, i) \leftarrow FALSE$.
  else $TRAVERSE(P, i+1)$,
    $TRAVERSE(P, i) \leftarrow TRUE$.

After the search algorithm for searching discrete dynamic networks has given a (legal and optimal) solution path P, unnecessary train changes can be eliminated by calling $TRAVERSE(P, 0)$.

11.5. Suboptimal solutions with fewer train changes

Until now we have been solely interested in optimal solutions, i.e. solutions with the least travel time regardless of the number of train changes. In a practical situation however, people often want to know not only the quickest solution, but also solutions which may take more time but have fewer train changes. In the following sections we shall look at different, increasingly complex cases of suboptimal solutions with fewer train changes. We shall show how the DYNET algorithm for searching discrete dynamic networks can be adapted to find some of these cases.

11.5.1. Competing solutions

A simple case of suboptimal solutions with fewer train changes is the case in which two solutions with different characteristics (in terms of travel time and train changes) compete at a vertex. For instance, consider the following example, in which we want to travel from Hk to Asd (see fig. 11.1). The changing time at Utg is 3 minutes.

	100	200	300	Station
Hk	8:00		8:00	Heemskerk
Utg	8:05	8:08	|	Uitgeest
Hlm		|	8:25	Haarlem
Zd		8:25	|	Zaandam
Ass		8:35	8:40	Amsterdam Sloterdijk
Asd		8:40	8:45	Amsterdam Central Station

The optimal solution, with one train change, is to travel from Hk to Utg by train 100, and from Utg to Asd by train 200. However, we could also travel from Hk to Asd directly by train 300, with only 5 minutes more travel time.

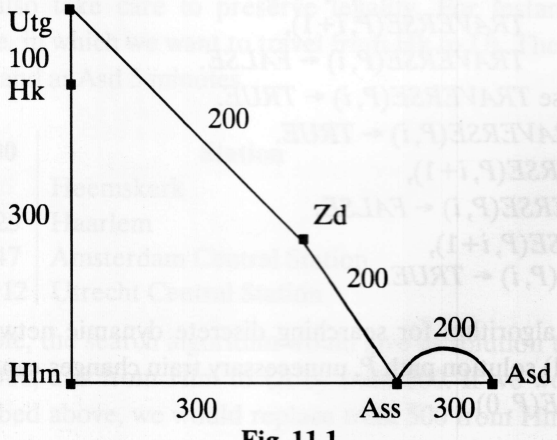


Fig. 11.1.

11.5.2. Using a change value

When we are using the DYNET algorithm in the example of fig. 11.1, from Hk Utg gets labeled 8:05 and Hlm gets labeled 8:25. Then Utg becomes the branching vertex and Zd gets labeled 8:25. Hlm becomes the branching vertex next and Ass gets labeled 8:40. Then Zd becomes the branching vertex and Ass gets relabeled 8:35. Ass becomes the branching vertex (by the partial path via Zd) and Asd gets labeled 8:40. When Ass becomes the branching vertex for the second time (by the path via Hlm) it is rejected because it arrives outside the *maxiCON* interval. So, the direct path is developed until Ass, and is then rejected, favouring the faster path.

We can find such suboptimal solutions with fewer train changes, which compete with optimal solutions with more train changes, by introducing a *change value*. The change value is the extra time we are prepared to travel to avoid one train change. It is the time a train change is "worth". In the algorithm for searching discrete dynamic networks, the *end* value of a path is used to evaluate a path in the forward pass, and the *start* value of a path is used in the backward pass. With the change value, we can give each path a *corrected end* value in the forward pass, and a *corrected start* value in the backward pass. The corrected *end* value of a path P, $c_{end}(P)$, is defined as:

$$c_{end}(P) = end(P) + CHANGES(P) * change\_value$$

where $CHANGES(P)$ is the number of train changes occurring in P.

Similarly, for the backward pass of the algorithm, the corrected *start* value is defined as:

$$c_{start}(P) = start(P) - CHANGES(P) * change\_value.$$

By having the algorithm search for paths with optimal corrected *start* and *end* values, whenever there is a choice, a suboptimal solution with fewer train changes is preferred.

We now give a formal definition of the DYNET algorithm, using corrected *start* and *end* values (for a detailed discussion of the different steps of the algorithm, please refer to chapter 6).

Consider a discrete dynamic network consisting of the graph $G = (V, E)$ and the connection cost function CON. The maximum value of CON at a vertex u is $maxiCON(u)$, which is non-zero. The number of train changes in a path P is denoted by $CHANGES(P)$. The *change_value* is the time that we are prepared to travel to avoid one train change. The corrected *end* value of a path P, $c_{end}(P)$, and the

corrected *start* value of a path P, $c_{start}(P)$, are defined as above. The two special vertices s and t of the network are the *starting* and *terminating* vertices. We want to find a legal path from s to t in our discrete dynamic network, where the corrected *end* value of the path is minimum and given this corrected *end* value, the corrected *start* value of the path is maximum and its *start* value at least T_{start}.

Pass 1:

(1) $\lambda(s) \leftarrow T_{start}$ and for all $v \in V, v \neq s$, $\lambda(v) \leftarrow \infty$.
Create a partial path P_0 consisting of s only, $c_{end}(P_0) \leftarrow T_{start}$.
For all $v \in V$, $\omega(v, u) = \infty$ for each neighbour u of v.
(2) $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j$ in F for which $c_{end}(P_m)$ is minimum; if F is empty then stop, no complete path could be found.
(4) if $u_j = t$, stop, P_m is a complete path with an optimal corrected end value.
(5) if $c_{end}(P_m) < \lambda(u_j) + maxiCON(u_j)$.
then for every relevant edge e_j : $u_j \rightarrow u_{j+1}$:
create a partial path $P_n = s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j, e_j, u_{j+1}$
$c_{end}(P_n) \leftarrow end(P_n) + CHANGES(P_n) * change\_value$
if $\lambda(u_{j+1}) > c_{end}(P_n)$
then $\lambda(u_{j+1}) \leftarrow c_{end}(P_n)$
if $c_{end}(P_n) < \lambda(u_{j+1}) + maxiCON(u_{j+1})$
then $F \leftarrow F + \{P_n\}$.
if $end(e_j) < \omega(u_{j+1}, u_j)$
then $\omega(u_{j+1}, u_j) \leftarrow end(e_j)$.
(6) $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_0, \ldots, u_{j-1}, e_{j-1}, u_j$ and a vertex u_{j+1}, then the relevant edges from u_j to u_{j+1} are the edges e_j: $u_j \rightarrow u_{j+1}$ for which the following two *ordered* conditions hold:

(1) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$, and
(2) $end(e_j) < end(e_{min}) + maxiCON(u_{j+1})$,
where $end(e_{min})$ is the minimum end value of any edge satisfying (1).

Pass 2:

(1) $\kappa(t) \leftarrow \lambda(t)$ and for all $v \in V, v \neq t$, $\kappa(v) \leftarrow -\infty$.
Create a partial path P_0 consisting of t only, $c_{start}(P_0) \leftarrow \lambda(t)$.
(2) $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $u_j, \ldots, u_{k-1}, e_{k-1}, t$ in F for which $c_{start}(P_m)$ is maximum.

(4) if $u_j = s$, stop, P_m is an optimal complete path.
(5) if $c_{start}(P_m) > \kappa(u_j) - maxiCON(u_j)$
then for every relevant edge e_{j-1} : $u_{j-1} \rightarrow u_j$, with $\omega(u_j, u_{j-1}) \leq start(P_m)$,
create a partial path $P_n = u_{j-1}, e_{j-1}, u_j, ..., u_{k-1}, e_{k-1}, t$
$c_{start}(P_n) \leftarrow start(P_n) - CHANGES(P_n) * change\_value$.
if $\kappa(u_{j-1}) < c_{start}(P_n)$
then $\kappa(u_{j-1}) \leftarrow c_{start}(P_n)$
if $c_{start}(P_n) > \kappa(u_{j-1}) - maxiCON(u_{j-1})$
then $F \leftarrow F + \{P_n\}$.
(6) $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_j, e_j, u_{j+1}, ..., u_k$ and a vertex u_{j-1}, then the relevant edges from u_{j-1} to u_j are the edges e_{j-1}: $u_{j-1} \rightarrow u_j$ for which the following two *ordered* conditions hold:

(1) $end(e_{j-1}) \leq start(e_j) - CON(u_j, e_{j-1}, e_j)$, and
(2) $start(e_{j-1}) > start(e_{max}) - maxiCON(u_{j-1})$,
where $start(e_{max})$ is the maximum start value of any edge satisfying (1).

Suppose that in the example of fig 11.1, the change value is 15. Then, in the forward pass, the corrected *end* value of the partial path Hk→Utg→Ass (with one train change) would be 8:35 + 1 * 15 = 8:50, and the corrected *end* value of the (direct) partial path Hk→Hlm→Ass would be 8:40 + 0 * 15 = 8:40. So, the direct path would be favoured.

11.5.3. Using macro operators

There are situations however, when the above approach will fail to find suboptimal solutions with fewer train changes. For example consider the following example in which we want to travel from Utg to Asd (see fig. 11.2). The changing time at Ass is 5 minutes.

	400	410	500	Station
Utg	8:00		8:00	Uitgeest
Hlm	|		8:25	Haarlem
Ass	8:30	8:35	8:40	Amsterdam Sloterdijk
Asd		8:40	8:45	Amsterdam Central Station

The optimal solution, with one train change, is to travel from Utg to Ass by train 400, and from Ass to Asd by train 410. However, we could also travel from Utg to Asd directly by train 500, with only 5 minutes more travel time. The algorithm described above will fail to find this solution. From Utg, Hlm is labeled 8:25 and Ass

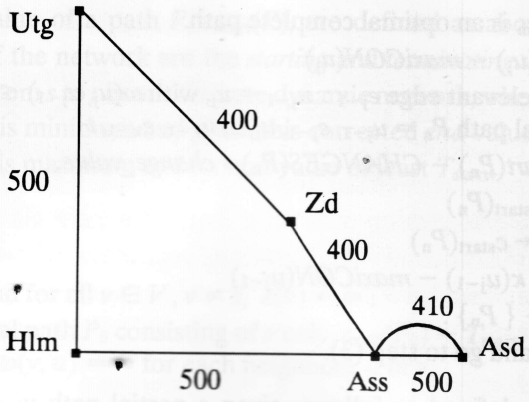


Fig. 11.2.

is labeled 8:30. Hlm becomes the next branching vertex and from Hlm, Ass is **not** relabeled and the new path (which would eventually result in the direct path) is not put in the frontier because it arrives outside the (corrected) *maxiCON* interval: the corrected *end* value of the partial path Utg→Ass is 8:30 + 0 * 15 = 8:30 (remember that the *change_value* is 15), while the corrected *end* value of the partial path Utg→Hlm→Ass is be 8:40 + 0 * 15 = 8:40. So, the path resulting in one train change is still preferred.

The problem is that, as a result of the path so far (from Utg to Ass), a train change is induced *later* (from Ass to Asd). So, we are faced with the problem that in order to find paths which will eventually result in suboptimal paths with fewer train changes, we must remember and develop more paths than those which arrive within the *maxiCON* interval only. However if we do not insist on finding all suboptimal paths with fewer train changes, we do not need to remember all paths, and those which we do need to remember need not be developed further in all directions.

11.5.3.1. Which paths to remember

Suppose the path with the best corrected *end* value which arrives at a vertex v, path P, has an *end* value of t, has CHANGES(P) changes, and a corrected *end* value of $c_{end}(P)$. We need to remember all paths which arrive at v and have the same number of train changes.

Paths with fewer train changes but a higher corrected *end* value have lost more time than we valued the decrease in train changes was worth, compared to P. Although it is not impossible that these paths "improve" later on (i.e. do become interesting because the number of train changes in the optimal solution has increased) and result in an interesting solution, we do not develop such paths

further. We have to make a trade-off between efficiency of search and the number of interesting solutions which we try to find.

Similarly, for paths with more train changes and a higher corrected *end* value, the excess number of train changes is not worth the decrease in travel time, compared to P. Again, there is no guarantee that these paths do not improve later on and result in an interesting solution, but we do not develop those paths further. At the end of this chapter we shall look into the possibility of paths which are not interesting locally but improve later on.

11.5.3.2. Which paths to develop

First we make the observation that, since all paths we remember have the same number of train changes and since all those paths have greater corrected *end* value than P, it must be that for all paths P_L which we remember, we have $end(P) \leq end(P_L)$. For the reasons describe in chapter 6, of these paths, those which arrive within the interval $end(P) + maxiCON(v)$ need to be developed one step further in all directions (i.e. allowing new train changes to occur). The other paths are only interesting if they result in fewer train changes later on. Since any train which can be used to change to from these paths, can also be used from path P, changing trains from these paths will not give interesting solutions which cannot be constructed from P. Therefore, we only develop the paths arriving outside the *maxiCON* interval using the train represented by the last edge, i.e. along its macro operator.

11.5.3.3. The DYNET algorithm, using macro operators

We now give a formal definition of the DYNET algorithm for searching discrete dynamic networks adapted to use macro operators to favour solutions with fewer train changes. In the algorithm, the number of the train changes of the (corrected) best path arriving at a vertex v is denoted by $\varphi(v)$.

Consider a discrete dynamic network consisting of the graph $G = (V, E)$ and the connection cost function *CON*. The maximum value of *CON* at a vertex u is $maxiCON(u)$. The number of train changes in a path P is denoted by $CHANGES(P)$. The *change_value* is the time that we are prepared to travel to avoid one train change. The corrected *end* value of a path P, $c_{end}(P)$, and the corrected *start* value of a path P, $c_{start}(P)$, are defined as previously. The two special vertices s and t of the network are the *starting* and *terminating* vertices. We want to find a legal path from s to t in our discrete dynamic network, where the corrected *end* value of the path is minimum and given this corrected *end* value, the corrected *start* value of the path is maximum and its *start* value at least T_{start}.

Pass 1:

(1) $\lambda(s) \leftarrow T_{start}$ and for all $v \in V, v \neq s,\ \lambda(v) \leftarrow \infty$.
Create a partial path P_0 consisting of s only, $c_{end}(P_0) \leftarrow T_{start}$.
For all $v \in V, \omega(v, u) = \infty$ for each neighbour u of v.
(2) $F \leftarrow \{P_0\}$.
(3) Let P_m be a partial path $s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j$ in F for which $c_{end}(P_m)$ is minimum; if F is empty then stop, no complete path could be found.
(4) if $u_j = t$, stop, P_m is a complete path with an optimal corrected end value.
(5) if $CHANGES(P_m) = \varphi(u_j)$ and $c_{end}(P_m) < \lambda(u_j) + maxiCON(u_j)$,
then for every relevant edge $e_j : u_j \rightarrow u_{j+1}$:
create a partial path $P_n = s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j, e_j, u_{j+1}$.
$c_{end}(P_n) \leftarrow end(P_n) + CHANGES(P_n) * change\_value$
if $\lambda(u_{j+1}) > c_{end}(P_n)$
then $\lambda(u_{j+1}) \leftarrow c_{end}(P_n)$ and $\varphi(u_{j+1}) \leftarrow CHANGES(P_n)$.
$F \leftarrow F + \{P_n\}$.
if $end(e_j) < \omega(u_{j+1}, u_j)$
then $\omega(u_{j+1}, u_j) \leftarrow end(e_j)$.
(6) if $CHANGES(P_m) = \varphi(u_j)$ and $\lambda(u_j) + maxiCON(u_j) \leq c_{end}(P_m)$.
then for the edge $e_j : u_j \rightarrow u_{j+1}$ for which $id(e_j) = id(e_{j-1})$:
create a partial path $P_n = s, e_0, u_1, \ldots, u_{j-1}, e_{j-1}, u_j, e_j, u_{j+1}$.
$c_{end}(P_n) \leftarrow end(P_n) + CHANGES(P_n) * change\_value$
if $\lambda(u_{j+1}) > c_{end}(P_n)$
then $\lambda(u_{j+1}) \leftarrow c_{end}(P_n)$ and $\varphi(u_{j+1}) \leftarrow CHANGES(P_n)$.
$F \leftarrow F + \{P_n\}$.
(7) $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_0, \ldots, u_{j-1}, e_{j-1}, u_j$ and a vertex u_{j+1}, then the relevant edges from u_j to u_{j+1} are the edges e_j: $u_j \rightarrow u_{j+1}$ for which the following two *ordered* conditions hold:

(1) $start(e_j) \geq end(e_{j-1}) + CON(u_j, e_{j-1}, e_j)$, and
(2) $end(e_j) < end(e_{min}) + maxiCON(u_{j+1})$,
where $end(e_{min})$ is the minimum end value of any edge satisfying (1).

Pass 2:

(1) $\kappa(t) \leftarrow \lambda(t)$ and for all $v \in V, v \neq t,\ \kappa(v) \leftarrow -\infty$.
Create a partial path P_0 consisting of t only, $c_{start}(P_0) \leftarrow \lambda(t)$.
(2) $F \leftarrow \{P_0\}$.

(3) Let P_m be a partial path $u_j,..., u_{k-1}, e_{k-1}, t$ in F for which $c_{start}(P_m)$ is maximum.
(4) if $u_j = s$, stop, P_m is an optimal complete path.
(5) if $\varphi(u_j) = CHANGES(P_m)$ and $c_{start}(P_m) > \kappa(u_j) - maxiCON(u_j)$,
then for every relevant edge $e_{j-1}: u_{j-1} \rightarrow u_j$, with $\omega(u_j, u_{j-1}) \leq start(P_m)$,
create a partial path $P_n = u_{j-1}, e_{j-1}, u_j,..., u_{k-1}, e_{k-1}, t$,
$c_{start}(P_n) \leftarrow start(P_n) - CHANGES(P_n) * change\_value$.
if $\kappa(u_{j-1}) < c_{start}(P_n)$
then $\kappa(u_{j-1}) \leftarrow c_{start}(P_n)$ and $\varphi(u_{j-1}) \leftarrow CHANGES(P_n)$.
$F \leftarrow F + \{P_n\}$.
(6) if $\varphi(u_j) = CHANGES(P_m)$ and $\kappa(u_j) - maxiCON(u_j) > c_{start}(P_m)$
then for the edge $e_{j-1}: u_{j-1} \rightarrow u_j$, with $id(e_{j-1}) = id(e_j)$,
and with $\omega(u_j, u_{j-1}) \leq start(P_m)$,
create a partial path $P_n = u_{j-1}, e_{j-1}, u_j,..., u_{k-1}, e_{k-1}, t$.
$c_{start}(P_n) \leftarrow start(P_n) - CHANGES(P_n) * change\_value$.
if $\kappa(u_{j-1}) < c_{start}(P_n)$
then $\kappa(u_{j-1}) \leftarrow c_{start}(P_n)$ and $\varphi(u_{j-1}) \leftarrow CHANGES(P_n)$.
$F \leftarrow F + \{P_n\}$.
(7) $F \leftarrow F - \{P_m\}$ and go to step (3).

A relevant edge is defined as follows: given a partial path $u_j, e_j, u_{j+1},..., u_k$ and a vertex u_{j-1}, then the relevant edges from u_{j-1} to u_j are the edges $e_{j-1}: u_{j-1} \rightarrow u_j$ for which the following two *ordered* conditions hold:

(1) $end(e_{j-1}) \leq start(e_j) - CON(u_j, e_{j-1}, e_j)$, and
(2) $start(e_{j-1}) > start(e_{max}) - maxiCON(u_{j-1})$,
where $start(e_{max})$ is the maximum start value of any edge satisfying (1).

11.5.3.4. Other cases

As we already mentioned, there are still cases in which the algorithm described above fails to find an interesting suboptimal solution. For instance, consider the following case in which we want to travel from Uitgeest to Utrecht CS (see fig. 11.3). The changing time at Hlm is 4 minutes and at Ass and Asd 5 minutes. The *change_value* is 15.

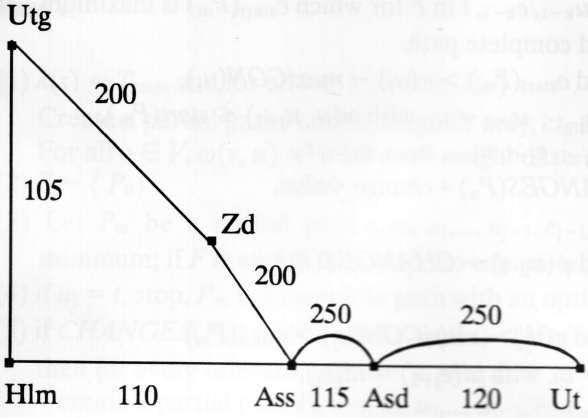


Fig. 11.3.

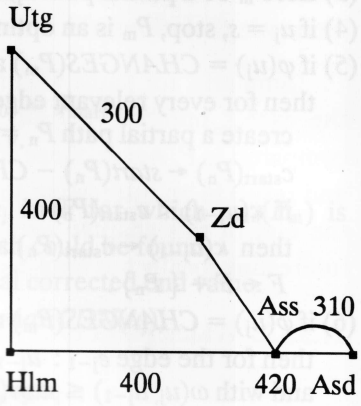


Fig. 11.4.

	105	110	115	120	200	250	Station
Utg	8:00				8:05		Uitgeest
Hlm	8:25	8:30			|		Haarlem
Zd		|			8:45		Zaandam
Ass		8:45	8:50		9:05	9:10	Amsterdam Sloterdijk
Asd			8:55	9:00		9:15	Amsterdam Central Station
Ut				9:25		9:40	Utrecht Central Station

The partial path Utg→Hlm→Ass has an *end* value of 8:45, with one train change, giving a corrected *end* value of 9:00. The partial path Utg→Zd→Ass has an *end* value of 9:05, with no train changes, so its corrected *end* value is 9:05. Since the extra 20 minutes is not worth the fewer train change, the path is not developed further. So, we do not find the path to Ut with an *end* value of 9:40, and a corrected *end* value of 9:55. The path we do find, arrives at Ut at 9:25 with three train changes, giving a corrected *end* value of 10:10. In order to find such a path, which has fewer train changes but have lost too much time locally and "improve" later on, we must develop paths with fewer train changes allowing new train changes to occur (i.e. not only develop it along its macro operator).

A similar situation can occur with solutions which have a better *end* value but which have locally too many train changes, and improve later on, compared to the optimal solution. For example consider the following case, in which we want to travel from Uitgeest to Amsterdam CS (see fig. 11.4). The changing time at Zd and Ass is 5 minutes. The *change_value* is 15.

	300	305	310	400	410	Station
Utg	8:00			8:00		Uitgeest
Hlm	|			8:25		Haarlem
Zd	8:15	8:20		|		Zaandam
Ass		8:30	8:35	8:40	8:55	Amsterdam Sloterdijk
Asd			8:40		9:00	Amsterdam Central Station

The partial path Utg→Hlm→Ass has an *end* value of 8:40, with no train change, so its corrected *end* value is 8:40. The partial path Utg→Zd→Ass has an *end* value of 8:30, with one train change, giving a corrected *end* value of 8:45. Since the extra train change is not worth the 10 minutes decrease in travel time, it is not developed further. So, we do not find the path to Asd with an *end* value of 8:40, and a corrected *end* value of 9:10. The path we do find, arrives at Ut at 9:00, giving a corrected *end* value of 9:15. In order to find these paths, we need to develop paths which have more train changes further, allowing new train changes to occur.

A final example is a situation in which a later non-relevant edge (train) results in a better path. Suppose we want to travel from Utg to Asd. The changing time at Hlm is 4 minutes, the changing time at Ass is 5 minutes. The *change_value* is 15.

	100	105	110	200	Station
Utg	8:00				Uitgeest
Hlm	8:10	8:15		8:30	Haarlem
Ass		8:30	8:35	8:45	Amsterdam Sloterdijk
Asd			8:45	8:55	Amsterdam Central Station

From the partial path Utg→Hlm (with train 100), train 105 is relevant and train 200 is not relevant. However, using train 200 would give a path with a corrected *end* value of 9:10 (the *end* value of the path would be 8:55, the one train change would give the path a corrected *end* value of 9:10). The path we do find (with train 100, 105 and 110) arrives at 8:45, and its two train changes give it a corrected *end* value of 9:15. In order to find such a path with an optimal corrected *end* value, we must develop all later non-relevant edges, not allowing new train changes (i.e. along their macro operator).

12. Implementation

In this chapter we shall discuss two implementation issues which both greatly influence the memory requirements and the (computational) speed of the algorithms discussed in the previous chapters. First we shall look at how a network can be represented in computer memory. Then we shall look at how the *frontier* (the collection F holding all tentatively labeled vertices) can be represented and handled when implementing a label setting algorithm (see chapter 3 for a discussion of label setting algorithms).

12.1. Representing a network

A network can be represented in computer memory in a number of ways. Without considering the actual search algorithm as such, an efficient representation of the network can speed up searching substantially. We distinguish four kinds of representation:

(1) matrix representation;
(2) ladder representation;
(3) forward star representation;
(4) sorted forward star representation.

In our example representations we shall use the network of fig. 12.1.

12.1.1. Matrix representation

In a matrix representation, the network is represented by a $|V| * |V|$ matrix. The matrix entry (i, j) contains the length of the edge connecting vertex i and vertex j if it exists; if the edge does not exists it contains ∞. The space requirement of this representation is $|V|^2$ times the space needed to store the attributes of an edge (the name and the length of an edge in a normal graph, the name and the *start* and *end* value of an edge in case of a discrete network). When the network is sparse, namely when the number of edges is significantly smaller than $|V|^2$, the matrix contains ∞'s for the larger part. Note that it is not possible to represent parallel edges in a normal,

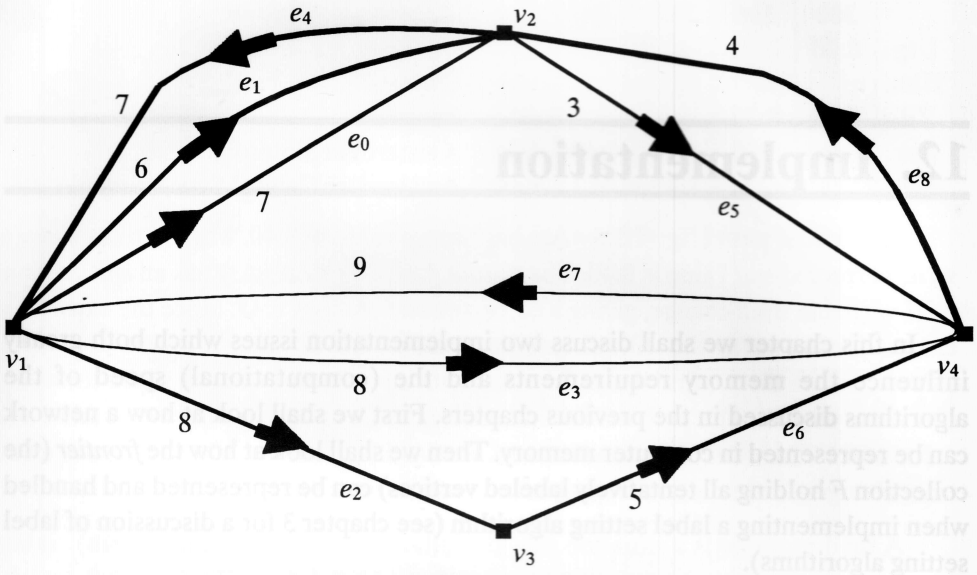


Fig. 12.1.

two dimensional matrix. If parallel edges occur, as in railway service networks, the matrix must either be made three dimensional or extra rows and columns must be added. The matrix representation of our example network looks as follows (an extra column for v_2 is added to store the parallel edges from v_1 to v_2):

	v_1		v_2		v_2		v_3		v_4	
v_1		∞	e_0	7	e_1	6	e_2	8	e_3	8
v_2	e_4	7		∞		∞		∞	e_5	3
v_3		∞		∞		∞		∞	e_6	5
v_4	e_7	9	e_8	4		∞		∞		∞

12.1.2. Ladder representation

In a ladder representation, all edges of the network are stored in a list, each entry consisting of the start vertex, the end vertex, the name and the length of an edge (or the *start* and *end* values of an edge in a discrete network). The list can be sorted on start vertex. The representation requires $|E|$ times the space required to store one entry (the two vertices and the attributes of an edge). Obviously, the list is always completely filled. The start vertex is stored redundantly when many edges depart from a vertex. The ladder representation of our example network looks as follows:

v_1	v_2	e_0	7
v_1	v_2	e_1	6
v_1	v_3	e_2	8
v_1	v_4	e_3	8
v_2	v_1	e_4	7
v_2	v_4	e_5	3
v_3	v_4	e_6	5
v_4	v_1	e_7	9
v_4	v_2	e_8	4

12.1.3. Forward star representation

In a forward star representation the edges in the ladder are ordered on their start vertex, which is stored non-redundantly. An array and a list are used in this representation. The array is a pointer array of length $|V|$. The list is a linear list of length $|E|$, describing the edges. Each entry contains the end vertex and the attributes of the edge. The elements of the pointer array indicate where in the list the edges departing from a vertex are stored. Apart from some gain in memory space, the advantage of this representation is the simplified searching procedure that can be used. In our algorithms we repeatedly have to determine the edges connecting the adjacent vertices of a particular vertex. The pointer array allows an efficient determination of these edges, without having to go through a (possibly sorted) list. The collection of adjacent vertices is called the forward star of a vertex. The forward star representation of our example network looks as follows:

$v_1 \rightarrow$	v_2	e_0	7
	v_2	e_1	6
	v_3	e_2	8
	v_4	e_3	8
$v_2 \rightarrow$	v_1	e_4	7
	v_4	e_5	3
$v_3 \rightarrow$	v_4	e_6	5
$v_4 \rightarrow$	v_1	e_7	9
	v_2	e_8	4

Note that, when many parallel edges occur (as in railway service networks), the end vertex is stored redundantly. This can be avoided by using the same approach for the storage of the end vertices as we used for the start vertices, and use a second pointer array. Such a representation of our example network looks as follows:

$v_1 \rightarrow$	$v_2 \rightarrow$	e_0	7
		e_1	6
	$v_3 \rightarrow$	e_2	8
	$v_4 \rightarrow$	e_3	8
$v_2 \rightarrow$	$v_1 \rightarrow$	e_4	7
	$v_4 \rightarrow$	e_5	3
$v_3 \rightarrow$	$v_4 \rightarrow$	e_6	5
$v_4 \rightarrow$	$v_1 \rightarrow$	e_7	9
	$v_2 \rightarrow$	e_8	4

12.1.4. Sorted forward star

In a sorted forward star representation, all end vertices in the forward star of a vertex are sorted on increasing edge length. The sorted forward star representation of our example network looks as follows:

$v_1 \rightarrow$	v_2	e_1	6
	v_2	e_0	7
	v_3	e_2	8
	v_4	e_3	8
$v_2 \rightarrow$	v_4	e_5	3
	v_1	e_4	7
$v_3 \rightarrow$	v_4	e_6	5
$v_4 \rightarrow$	v_2	e_8	4
	v_1	e_1	9

In the *Dantzig* implementation of a label setting algorithm (see [Da, 1986]), when a vertex is made permanent, only the nearest vertex in the forward star of this vertex is visited. Because the nearest vertex of the forward star will get a smaller label than the other adjacent vertices, it will also be the first of these vertices to become permanent. When this vertex then actually becomes permanent, however, because this vertex was the only vertex that was visited from the forward star of its predecessor, the predecessor has to be checked again to determine the next vertex from the forward star to be visited (from the predecessor).

For an example, suppose that in fig. 12.1, we search for a path from v_1 to v_4. When v_1 becomes permanent, only the nearest vertex in the forward star of v_1, v_2, is visited and labeled 6. Then v_2 becomes permanent. Since v_2 was the only vertex from the forward star of v_1 that was visited, we must now determine the next vertex from the forward star of v_1: v_3, which is visited and labeled 8. From v_2, the first vertex from the forward star of v_2, v_4 is visited and labeled 9. Then v_3 becomes permanent. Since

v_3 was not the last vertex from the forward star of v_1, we determine the next vertex to be visited from v_1: v_4, which is visited and labeled 8. Furthermore, from v_3, the first vertex of its forward star, v_4 is visited and labeled 13. Then v_4 becomes permanent and we have found a shortest path. In this example, only one vertex from the forward star of v_2 and v_3 needed to be visited, whereas in a usual implementation all vertices in the forward star of these vertices would have been visited.

Of course, in a discrete dynamic network the edges can be sorted on shortest length and then on *start* value, however, since in a discrete network the distance to the next vertex not only depends on the length and the *start* value of an edge, but also on the **current label** of the source vertex, it is not possible to determine beforehand which vertex from the forward star will be nearest. So, if we want to use the Dantzig implementation in a discrete (dynamic) network, we must determine the nearest vertex in the forward star at run-time.

12.2. Implementing the frontier

Label setting algorithms select the vertex from the collection of tentatively labeled vertices (the frontier: the collection F in our descriptions of label setting algorithms, which include all algorithms we described for searching discrete and discrete dynamic networks), that has the smallest label (the currently shortest distance from the source vertex). Each vertex that is labeled is put in the frontier. So, in order to make access to the frontier efficient, it must be implemented in such a way that:

(1) the vertex with the lowest label can be determined easily;
(2) new elements can be readily inserted into the frontier.

We shall discuss five ways to implement the frontier:

(1) a sorted list;
(2) a binary heap;
(3) address calculation;
(4) circular address calculation;
(5) address calculation with buckets.

In our discussion we shall use the following example and assume integer values. Suppose that we are searching some network for a shortest path from the source vertex to the goal vertex, and that at some time during search, the frontier consists of the following vertices: v_0 with $\lambda(v_0) = 25$, v_1 with $\lambda(v_1) = 27$, v_2 with $\lambda(v_2) = 27$, v_3 with $\lambda(v_3) = 28$, v_4 with $\lambda(v_4) = 29$, v_5 with $\lambda(v_5) = 30$, v_6 with $\lambda(v_6) = 32$, v_7 with $\lambda(v_7) = 32$, v_8 with $\lambda(v_8) = 33$, v_9 with $\lambda(v_9) = 34$ (see for example the figure of

section 12.2.1). We shall show the frontier before the element with the smallest label (the vertex v_0) is determined, after it has been removed, and after the next element v_{10}, with $\lambda(v_{10}) = 31$, has been inserted.

12.2.1. Sorted list

A simple structure is to use a sorted list to represent the frontier. The list is sorted in ascending order with the vertex with the smallest label at the front. Inserting a new element into the frontier takes at most $|F|$ comparisons (where $|F|$ is the number of elements in the frontier). Removing the vertex with the smallest label does not cost any comparisons. Before the vertex with the smallest label is determined, our example looks as follows:

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
25	27	27	28	29	30	32	32	33	34

After the vertex with the smallest label (v_0) has been removed, the frontier looks as follows:

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
27	27	28	29	30	32	32	33	34

After vertex v_{10} with label 31 has been added, the frontier looks as follows:

v_1	v_2	v_3	v_4	v_5	v_{10}	v_6	v_7	v_8	v_9
27	27	28	29	30	31	32	32	33	34

12.2.2. Binary heap

A more efficient way of implementing the frontier is to use a binary heap. A binary heap is a binary tree structure in which an element (of the tree) at a higher level has a smaller label than all other elements in its two subtrees. The vertex with the smallest label is the top element of the tree. Fig. 12.2 shows how the frontier might look like before the element with the smallest label is removed.

When the element with the smallest label (the top element) is removed, its place is taken by the child element which has the smallest label of the two. This element in turn, is replaced by its child element with the smallest label, and so on, until the last level of the tree has been reached and the tree has been rearranged. Rearranging the tree in this way costs at most $\log_2|F|$ comparisons. In our example, the top element v_0 is replaced by v_1, v_1 by v_2, v_2 by v_8. The tree then looks as in fig. 12.3.

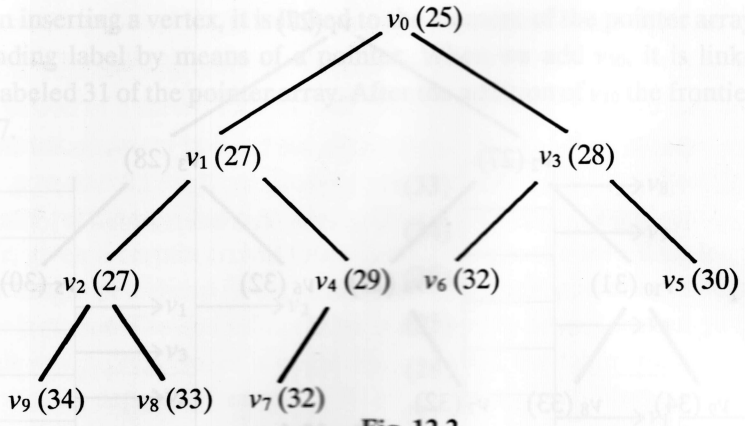


Fig. 12.2.

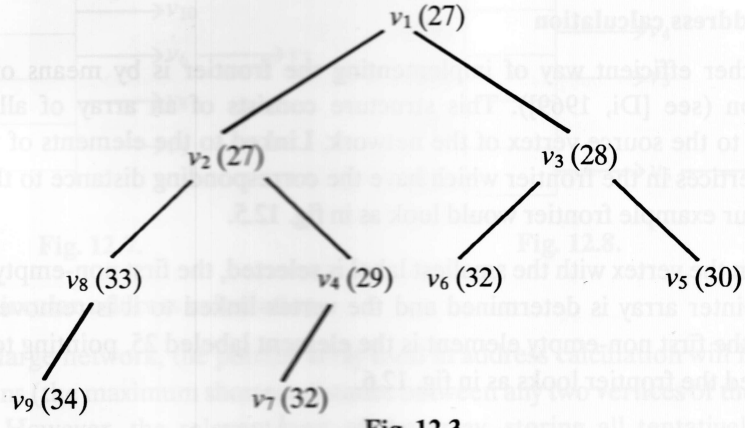


Fig. 12.3.

A new element is inserted at the first free leaf of the tree. Then the label of this element is compared to the label of its parent element. If the new element's label is smaller than the label of its parent, then the two elements are interchanged. Then its label is compared to the label of its new parent, and so on, until the label of the parent is smaller than the label of the new element (so there is no interchange necessary) or when the top of the tree has been reached. Rearranging the tree after inserting a new element takes at most $\log_2|F|$ comparisons. In our example, v_{10} is inserted as the second child of v_8. Since its label is smaller than the label of v_8, v_{10} and v_8 are interchanged. Since the new parent of v_8, v_2, has a smaller label than v_{10}, no next interchange is necessary, and the tree has been rearranged. The tree then looks as in fig. 12.4.

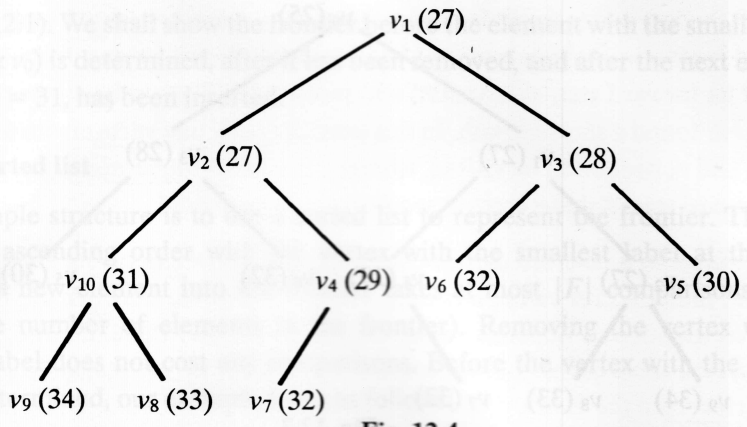


Fig. 12.4.

12.2.3. Address calculation

Another efficient way of implementing the frontier is by means of address calculation (see [Di, 1969]). This structure consists of an array of all possible distances to the source vertex of the network. Linked to the elements of this array are the vertices in the frontier which have the corresponding distance to the source vertex. Our example frontier would look as in fig. 12.5.

When the vertex with the smallest label is selected, the first non-empty element of the pointer array is determined and the vertex linked to it is removed. In our example the first non-empty element is the element labeled 25, pointing to v_0. With v_0 removed the frontier looks as in fig. 12.6.

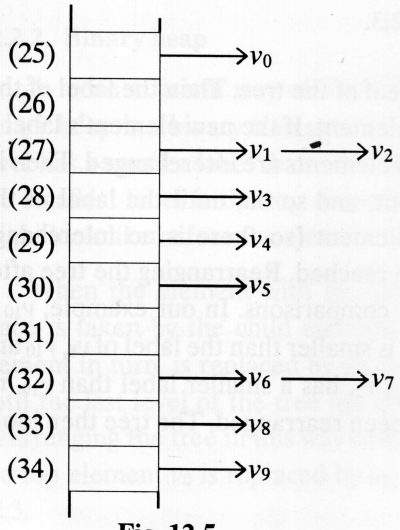


Fig. 12.5.

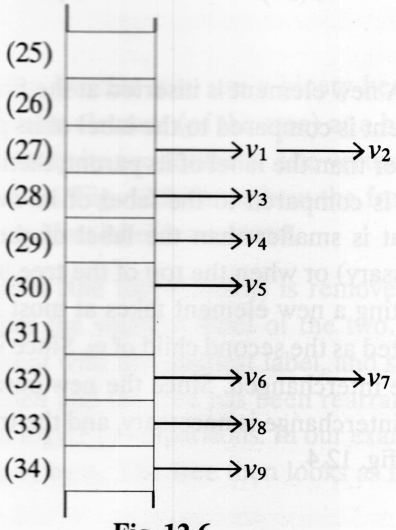


Fig. 12.6.

When inserting a vertex, it is linked to the element of the pointer array with the corresponding label by means of a pointer. When we add v_{10}, it is linked to the element labeled 31 of the pointer array. After the addition of v_{10} the frontier looks as in fig. 12.7.

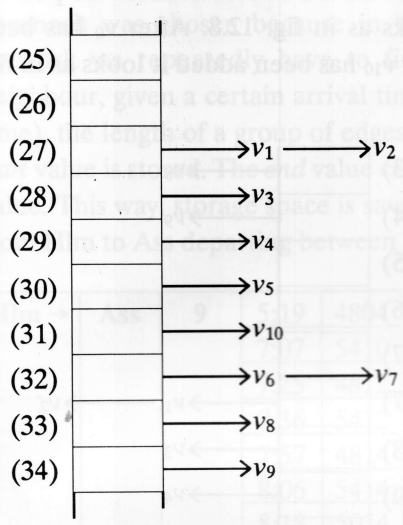


Fig. 12.7.

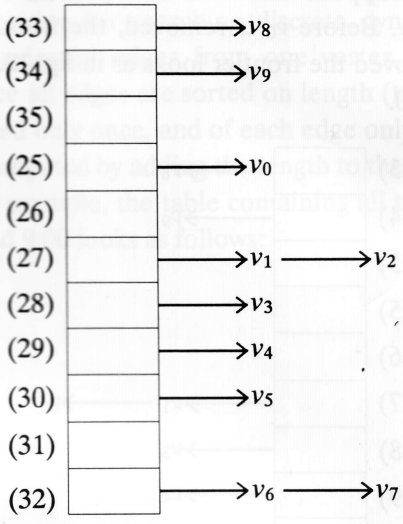


Fig. 12.8.

12.2.4. Circular address calculation

In a large network, the pointer array used in address calculation will have large dimensions (the maximum shortest distance between any two vertices of the network is long). However, the relevant part of the array, storing all tentatively labeled vertices, has only $l_{max} + 1$ positions, where l_{max} is the maximum length of any edge from the network. It is the maximum distance any neighbour can be from any vertex.

It is easily seen that the relevant part of the array is of length only $l_{max} + 1$, by the fact that in a label setting algorithm, the current vertex (the vertex that was made permanent most recently) is the vertex with the smallest label. No vertex with a larger label has been made permanent yet. So, no vertex which has not been made permanent yet, but which was visited from any of the vertices which were made permanent (and thus must be in the frontier), can have a label which is larger than the label of the current vertex plus the maximum distance any neighbour can be from any vertex. So, by using l_{max} elements, plus one element for the label of the current vertex (there may be multiple vertices with this label), we can limit the size of the pointer array to $l_{max} + 1$ elements. By subtracting the label of the current vertex from the label of the vertices which are visited, the address in the array of the new vertex

can be calculated. Note that in a discrete network, the maximum distance any neighbour can be from any vertex, is the maximum edge length **plus** the maximum wait time (the difference between the *start* and *end* values).

Suppose that in our example, l_{max} is 10. So, we need 11 elements for the pointer array. Before v_0 is removed, the frontier looks as in fig. 12.8. After v_0 has been removed the frontier looks as in fig. 12.9, after v_{10} has been added it looks as in fig. 12.10.

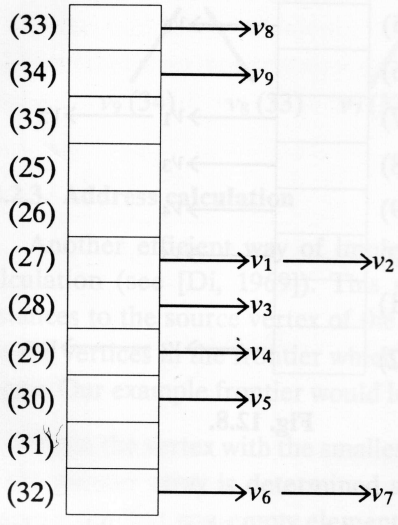


Fig. 12.9.

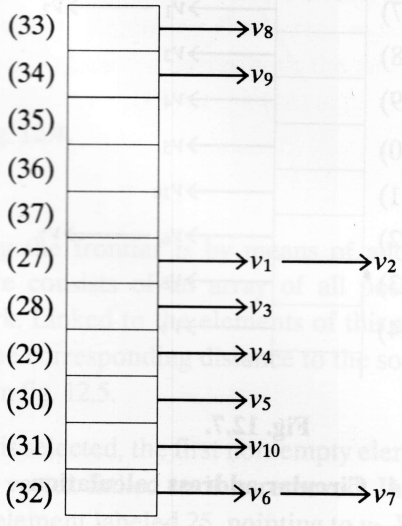


Fig. 12.10.

12.2.5. Address calculation with buckets

When the pointer array used in address calculation is very large (because of a large l_{max}), the array may be sparsely filled. In that case it may be time consuming to find the first non-empty element. It may then be useful to split the array into segments or buckets. A bucket is a range of elements of the pointer array. When any element in the range of a bucket is non-empty, than that bucket is marked non-empty. To indicate whether a bucket is empty or not we use an array, each element representing a bucket. When searching for the first non-empty element of the pointer array, we first search for the first non-empty bucket, and then for the first non-empty element in the range of the bucket. For the bucket structure, we need an extra array of length $(l_{max} + 1)/w$, where w is the range of the bucket (the bucket width).

12.3. Implementation of TRAINS

For the representation of the railway service network in TRAINS, we use an adapted version of a sorted forward star. The edges in the forward star are ordered on start vertex, then on end vertex, then on length, and then on *start* value. This approach was chosen because in the algorithm for searching discrete dynamic network we repeatedly have to find all relevant edges from one vertex to a neighbour, given a certain arrival time. Since all edges are sorted on length (travel time), the length of a group of edges is stored only once, and of each edge only the *start* value is stored. The *end* value can be computed by adding the length to the *start* value. This way, storage space is saved. For example, the table containing all trains from Hlm to Ass departing between 5:00 and 9:00 looks as follows:

Hlm →	**Ass**	**9**	5:19	4804
			7:07	5410
			7:25	4812
			7:36	5412
			7:57	4814
			8:06	5414
			8:18	5014
			8:25	4816
			8:37	5416
	Ass	**10**	6:42	923
			6:53	4810
			7:12	825
			7:18	5010
			7:41	927
			7:48	5012
			8:12	829
			8:42	931
			8:50	5016
			8:57	4818
	Ass	**11**	5:45	4806
	Ass	**13**	6:23	4808

First the departures and the identification numbers of the trains from Hlm to Ass with a travel time of 9 minutes are listed, then the trains from Hlm to Ass with a travel time of 10 minutes, then one train with a travel time of 11 minutes, and finally one with a travel time of 13 minutes.

For the implementation of the frontier we have chosen circular address calculation. Since elements are added to the frontier very frequently, an efficient

determination of its place in the frontier is important. For reasons of future applications including airline schedules, we have chosen a frontier size of 24:00 (which serves as an upper bound for the maximum travel and wait time for any connection).

13. TRAINS, An Active System

TRAINS is an implemented system which helps users find the fastest or most convenient train connections from any one station to any other station within The Netherlands. The entire Dutch railway services network is known to TRAINS, including special trains, connection times, restrictions, etc. TRAINS is currently being used at telephonic information centers of the Dutch railway company NS (Nederlandse Spoorwegen), and was recently introduced to the general public. Since TRAINS was intended to run on a personal computer, and had to be a complete system with a high performance, standard programming tools instead of tools such as fourth generation (database) languages or expert system shells were used to build TRAINS. Apart from a user friendly user-machine interface, TRAINS has an active component supplying useful alternate solutions in addition to the first and most apparent answer to the user's question. A "common sense" user model helps to select relevant information. The active component contributes significantly to the system, which is now highly valued by its users and well suited for everyday use. This chapter is largely similar to [Tu, 1989].

13.1. The theory of active systems: discontinuities

The theory of active systems was first described in [Si, 1978]. A good review of active systems can be found in [Wa, 1985]. The theory of active systems is based on discontinuities : a small change in the question may yield a large favourable change in the answer. It is easily seen that the domain of TRAINS, namely passenger transportation by train, exhibits discontinuities. It may be that a slow train with a travel time of 30 minutes departs at 10:00, while at 9:55 some very fast train with a travel time of 15 minutes leaves. So, a change of 5 minutes in the question (from a departure at 10:00 to a departure at 9:55) will give a change of 15 minutes in the answer (from 30 to 15 minutes in travel time). The metrics in both the question (what is "small") and the answer (what is "better") are user dependent.

13.2. The necessity of active behaviour

A user of a question-answering system seldom has a perfectly well-defined question, and seldom is this question definite. A precise question is posed because it is required. Indeed, we suspect that users usually overspecify their questions partly due to habit, partly due to requirements of the system. A system must have an exact value to work with, so the user gives some value which she thinks is a reasonable one. Given sufficient motivation, such as an answer which suits her better, she may change her question. Even if she chooses not to, the information should be provided so that the user can make a decision based on sufficient relevant information. In our implementation of TRAINS, we have assumed that the user does not know exactly when she wants to leave or arrive until she has seen the exact possibilities, which the system should provide.

13.3. The dimensions of a topic

The original definition of the theory of active systems states that a system should provide additional information when, and only when, a small change in the question results in a large positive change in the answer. But what is a small change in the question? And what is a positive change in the answer? And when is this change a large change? There may be many aspects of an answer which may influence the decision whether a change is positive or not, or whether it is a large or small improvement. These aspects are the attributes of the topic. Each of these attributes can be called a dimension of the topic. With this concept of a topic having multiple dimensions, a question and an answer can be viewed as points in a multi-dimensional space of possibilities.

In TRAINS, the topic is travel by train. The possible dimensions include the total travel time, the exact time of departure, the exact time of arrival, the number of train changes, the service available on the train (such as a dining car or the availability of a telephone), the route of the journey (the scenery) etc. In our implementation of TRAINS we limited the number of dimensions to the first four.

13.4. Active behaviour

In active behaviour we distinguish three phases:

(1) the subject focusing phase,
(2) the initial answer,
(3) searching for alternate solutions.

We shall now look at each phase in turn.

13.4.1. Subject focusing

When a user queries a question answering system, first she will formulate a question. In a conventional question answering system the question is treated as a precise definition of which information the user requires. In an active question answering system, the question is treated as a loose indication of what the user *wants to know*. In this phase, called the subject focusing phase, the user provides information which determines the most important dimensions of the question.

13.4.2. The initial answer

After the user has posed the question, the system starts searching for an answer best satisfying this initial question. This answer is given to the user. Usually this answer is the same answer as a conventional system would give. However, sometimes it may be best to avoid answering the initial question, and move directly to alternate solutions.

13.4.3. Searching for alternate solutions

After the answer to the initial question has been given, a conventional system stops. An active system starts "moving" the question along the most important dimensions, looking for favourable changes in answers. If a favourable change occurs, additional information is given to the user. Of course, this "moving" requires efficient search. Each slight move of the question along its dimensions results in a new question. This question needs to be answered internally in order to decide whether it yields a favourable change.

13.5. Discontinuity conflicts

It may be that a small change in the question results in large changes in multiple dimensions of the answer. Suppose large changes occur in two of the dimensions of the answer: one favourable and one unfavourable. For example, an alternate trip requires 15 minutes less travel time, but with two additional train changes. Is this a favourable change in answer? Should it be mentioned to the user? It is not possible to decide in general: it depends on the relative importance to the user of the dimensions travel time and number of train changes. Sometimes, some dimensions seem to be almost indistinguishable but there may be subtle differences in the metric, depending on the user. Consider for instance, the dimension travel time and the dimension time of arrival. The dimension travel time is easily measured and valued in terms of appreciation. Usually, less travel time is a positive change, more travel time is a negative change. But this is not the case with the dimension time of arrival. The level of appreciation depends on secondary conditions only known to

the user. If a user asks for a trip arriving at 9:55, an arrival at 10:05 may be unsuitable because she has a business appointment at 10:00. In that case an arrival at 9:40 may be the better possibility. However, if the goal of the trip is to go shopping, and if the shops do not open until 10:00, then the arrival at 10:05 could be perfectly acceptable.

13.6. User models

From the previous example it is seen that the metrics of the dimensions can be very much user dependent. For one user, a change in the answer may be an improvement, for another it may not, depending on the goal of the trip, or the level of experience in travelling by train (changing trains etc.), or even on the character or mood of the user. So, to be able to judge whether a change in the question is small, whether a change in the answer is large and favourable, and to be able to resolve discontinuity conflicts (in short: to judge whether an alternate solution is "better" or "near" the initial solution), we need a model of the user. There are two ways of dealing with user models:

(1) using predetermined models or user categories,
(2) building a model for each specific user.

Using predetermined user models means that during the subject focusing phase a user is classified into a category. We could have different categories for business travellers, students, elderly people, etc. When we build a user model for each specific user, for each new user a new user model is built in an interactive process.

13.7. The application of TRAINS

In the application of TRAINS, a user calls an NS telephone information center and asks for information about a trip from one station to another, giving a desired time of departure or arrival. Usually no additional information is supplied by the caller. There is a shortage of information, not enabling the system to classify the user or to build a user model. Of course, more information could be asked from the user but that would be undesirable. Most callers would not appreciate a complete questionnaire before their question is answered (the call is not even toll free!), and a user may become suspicious, feeling that her privacy is being invaded.

13.8. The user model in TRAINS

As we have seen in the previous section, either determining or building a user model before searching for a solution is not practical for our application. Another possibility might be to build the user model as we are going along. If a decision about what is "near" or "better" is necessary, a question could be put to the user. For

example, we could ask whether she favours fewer train changes or less travel time. But in that case, the user will probably reply that she cannot say in general and needs to know the exact possibilities to decide. A better approach would be to give the alternate solution right away, and ask the user whether it represents an improvement. From her answer, her (user) model could be modified. But then, we might just as well have given this alternate solution directly and let the user herself judge whether it is an improvement or not, without having to communicate her decision to the system. Of course, there is the danger of flooding her with many possible solutions, most undesirable! In TRAINS we have found a compromise: a rudimentary user model is implemented, incorporating some "common sense" about what might be an improvement to a user and what not. This rudimentary user model is sufficiently general to allow solutions which are of interest to essentially all users, and restrictive enough to prevent flooding her with possibilities. From the solutions suggested, she chooses the one which suits her best.

13.9. Relevant solutions

TRAINS' user model judges, for every alternate solution found, whether it may be relevant to a user or not. Relevant solutions are communicated to the user who decides which one is best for her. In this way, the system supplies everything likely to be necessary for the caller to make a decision with full knowledge of sufficient relevant information. A relevant solution is (recursively) defined as follows:

- The initial best solution is a relevant solution. If the user had given a desired departure time, the initial best solution makes her leave at or after this desired departure time and arrive at her destination as early as possible, with a trip of shortest duration (given that arrival time). Notice that in this way, she will leave as late as possible as long as she still arrives at the above earliest arrival time. (If the user had given a desired time of arrival, the initial best solution makes her depart as late as possible but still arrive before or at her desired arrival time, and given this departure time, will make her arrive as early as possible. Again, the journey has a minimal duration given the departure time). In case of ties, the route with the fewest train changes is preferred.
- Every solution with both the time of arrival and the time of departure different from those of another relevant solution is a relevant solution, provided that either:

  Both its departure and its arrival are earlier than the corresponding values of a relevant solution.

  or

  Both its departure and its arrival are later than the corresponding values of a relevant solution.

- Every solution with a departure equal to or earlier than, and/or an arrival equal to or later than the corresponding values of a relevant solution, is relevant if it has fewer train changes.

This way, first an optimal solution best satisfying the user's wishes is shown. Then trips are shown which have different times of departure and arrival and which are not just worse versions (in travel time) of relevant solutions already found. Finally, solutions which are worse versions (in travel time) of relevant solutions are shown if they have fewer train changes. These solutions exhibit discontinuity conflicts: more travel time but fewer train changes. It is left to the user to decide.

The interval that is searched for relevant solutions is determined by applying an heuristic formula taking into account such facts as the duration of the initial best solution, the amount of time the initial best solution differs from the user's question, etc. As a rule, always at least one alternate solution before and one after the initial best solution is given. To prevent too many alternate solutions, at most three time-different relevant solutions (see the second clause of the definition of relevant solutions) before and three after the initial best solution are given, and in addition, per initial best solution and time-different relevant solution, one non-time-different relevant solution with fewer train changes is allowed (see the third clause of the definition of relevant solutions).

13.10. An example

For an example (taken from the 1990/1991 NS service), suppose we want to travel from Hengelo (Hgl, see fig. 13.1) to Maastricht (Mt), departing at or after 9:00. There are four routes possible:

(1) via Almelo (Aml), Deventer (Dv), Amersfoort (Amf), Utrecht (Ut), 's Hertogenbosch (Ht), Eindhoven (Ehv), Roermond (Rm) and Sittard (Std), covering a (tariff) distance of 310 km,
(2) via Aml, Dv, Zutphen (Zp), Arnhem (Ah), Nijmegen (Nm), Venlo (Vl), Rm and Std, covering a (tariff) distance of 247 km,
(3) via Goor (Go), Zp, Ah, Nm, Vl, Rm and Std, covering a (tariff) distance of 223 km.
(4) via Go, Zp, Ah, Nm, Ht, Ehv, Rm and Std, covering a (tariff) distance of 266 km.

If our traveller wants to leave at 9:00, we look in the neighbourhood of 9:00 and find the following trip possibilities:

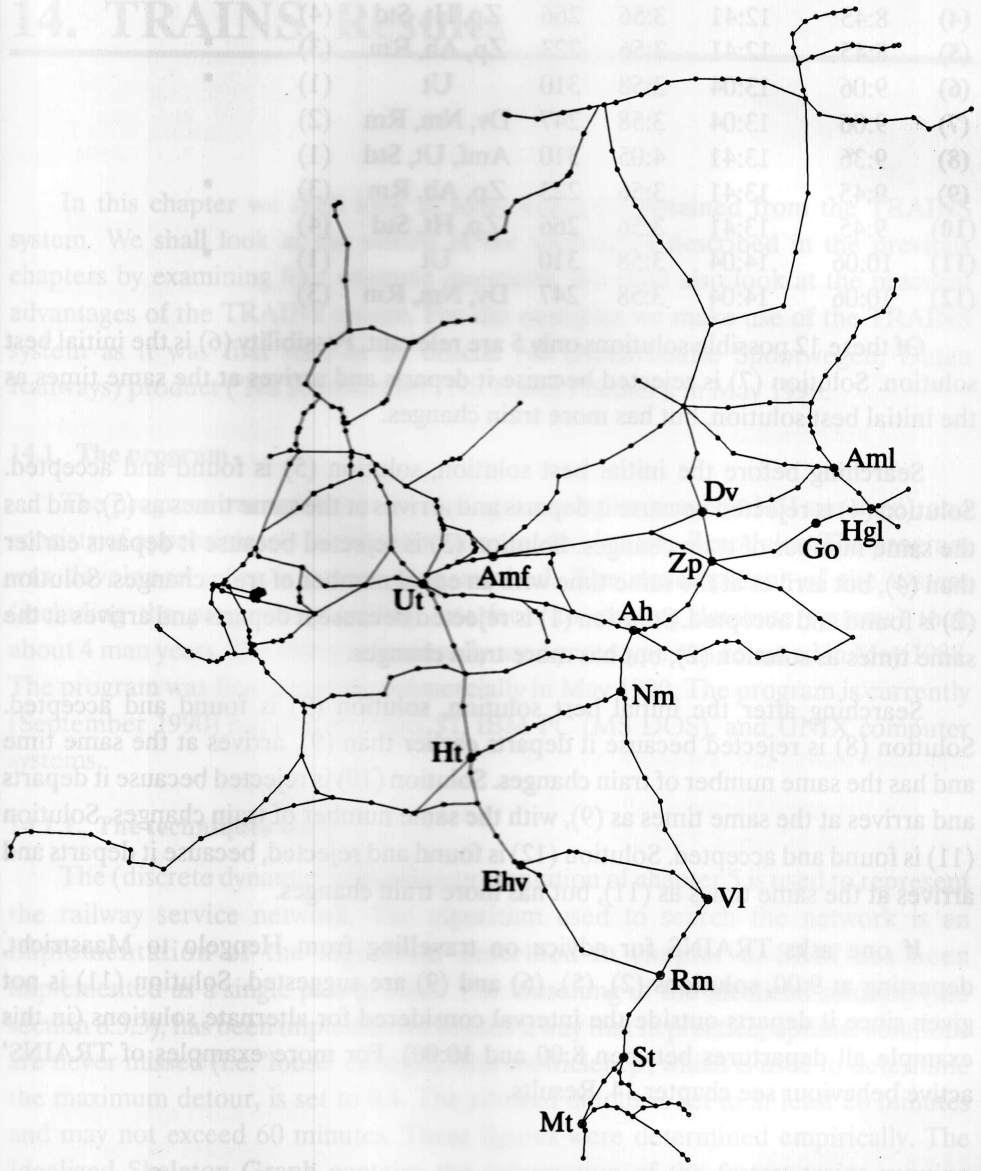


Fig. 13.1. The Dutch railway service network

	Departure	Arrival	Time	Dist.	Changes	Route	Relevant
(1)	8:06	12:04	3:58	247	Dv, Nm, Rm	(2)	
(2)	8:06	12:04	3:58	310	Ut	(1)	*
(3)	8:36	12:41	4:05	310	Amf, Ut, Std	(1)	
(4)	8:45	12:41	3:56	266	Zp, Ht, Std	(4)	
(5)	8:45	12:41	3:56	223	Zp, Ah, Rm	(3)	*
(6)	9:06	13:04	3:58	310	Ut	(1)	*
(7)	9:06	13:04	3:58	247	Dv, Nm, Rm	(2)	
(8)	9:36	13:41	4:05	310	Amf, Ut, Std	(1)	
(9)	9:45	13:41	3:56	223	Zp, Ah, Rm	(3)	*
(10)	9:45	13:41	3:56	266	Zp, Ht, Std	(4)	
(11)	10:06	14:04	3:58	310	Ut	(1)	*
(12)	10:06	14:04	3:58	247	Dv, Nm, Rm	(3)	

Of these 12 possible solutions only 5 are relevant. Possibility (6) is the initial best solution. Solution (7) is rejected because it departs and arrives at the same times as the initial best solution, but has more train changes.

Searching before the initial best solution, solution (5) is found and accepted. Solution (4) is rejected because it departs and arrives at the same times as (5), and has the same number of train changes. Solution (3) is rejected because it departs earlier than (4), but arrives at the same time with an equal number of train changes. Solution (2) is found and accepted. Solution (1) is rejected because it departs and arrives at the same times as solution (2), but has more train changes.

Searching after the initial best solution, solution (9) is found and accepted. Solution (8) is rejected because it departs earlier than (9), arrives at the same time and has the same number of train changes. Solution (10) is rejected because it departs and arrives at the same times as (9), with the same number of train changes. Solution (11) is found and accepted. Solution (12) is found and rejected, because it departs and arrives at the same times as (11), but has more train changes.

If one asks TRAINS for advice on travelling from Hengelo to Maastricht, departing at 9:00, solutions (2), (5), (6) and (9) are suggested. Solution (11) is not given since it departs outside the interval considered for alternate solutions (in this example all departures between 8:00 and 10:00). For more examples of TRAINS' active behaviour see chapter 14, Results.

14. TRAINS, Results

In this chapter we shall look at some examples obtained from the TRAINS system. We shall look at the effects of the techniques described in the previous chapters by examining four example questions. We shall also look at the practical advantages of the TRAINS system. For the examples we make use of the TRAINS system as it was first sold as an official NS (Nederlandse Spoorwegen; Dutch Railways) product ("NS Reisplanner", NS Travel Planner), in May 1990.

14.1. The program

The program was written in the C programming language (ANSI standard), and consists of approximately 10 000 lines of code, divided into 7 modules. The program was developed using a prototyping method. The development of the system (including the programs providing interfaces to existing database systems) took about 4 man years. The first professionally used prototype was released in May 1988. The program was first released commercially in May 1990. The program is currently (September 1990) running on Atari ST, IBM PC (MS DOS), and UNIX computer systems.

14.1.1. The techniques used

The (discrete dynamic) network representation of chapter 5 is used to represent the railway service network. The algorithm used to search the network is an implementation of the algorithm described in chapter 6. SRM has been implemented as a single pass process. The loosening of the idealized solution (see section 8.3.3), has been implemented in such a way that in practice, optimal solutions are never missed (i.e. 'loose' enough). The coefficient p, which is used to determine the maximum detour, is set to 0.4. The allowed detour is set to at least 20 minutes and may not exceed 60 minutes. These figures were determined empirically. The Idealized Skeleton Graph contains the information of the fastest trains running between the different stations. The ISG is constructed automatically and only once for the network and is stored together with the actual time-table information. The

ISG consists of about 450 stations and some 500 edges. In our implementation of SRM we do not cut off any dead-end branches in the search space. As a result of applying SRM, all stations in the search space have estimates of remaining travel time. These estimates are used in an A* extension (DYNET*) as described in chapter 9. The techniques to adapt the searching algorithm to offset vertices described in chapter 10, were also implemented. The determination of offset vertices is not automatic and is done by hand. The techniques described in chapter 11 are used to minimize train changes and to find suboptimal solutions with fewer train changes.

14.2. The network

The network contains all stations, trains, ferries and buses which are published in the official 1990 - 1991 NS time-tables (about 750 pages). Apart from all national trains, the network also contains the most important international trains to neighbouring countries, the most important ferries and some intercity bus links. In total the network has 469 stations: 396 Dutch stations (both railway stations and bus and ferry terminals) and 73 stations in neighbouring countries. Using the techniques described in chapter 10, 235 stations could be made offset stations (vertices) and 234 stations had to be node stations. The network contains the information of over 44 000 departures. Only 27 000 departures are departures from node stations. Binary coding techniques are used to store the network using the representation techniques described in chapter 12. The binary representation of the network requires a storage space of 251 820 bytes. At run-time, the entire network is kept in core memory.

14.3. The example questions

For the examples we shall look at four queries: one short trip, one medium trip and two longer trips. The short trip is from Heemskerk (Hk, see fig. 14.1) to Amsterdam CS (Asd), the medium trip from Hoorn (Hn) to Den Haag HS (Gv), the longer trips from Den Haag CS (Gvc) to Blerick (Br) and from Vlissingen (Vs) to Zwolle (Zl). For each query we shall look at the answers the system generates, the estimates and search space which are determined using SRM, and how the answers can be found using the conventional (paper) time-tables. The performance aspects and the computational effects of SRM and DYNET* are discussed in a separate section for all examples.

14.3.1. Heemskerk to Amsterdam CS

For a trip from Heemskerk to Amsterdam CS, the estimated travel time determined in SRM is 25 minutes. The upper bound is determined to be 45 minutes:

the estimated travel time of 25 minutes plus the minimum detour of 20 minutes. The search space consists of 13 node stations, 5.5 percent of the total network. The search space is given in fig. 14.2, including the relevant offset stations.

For a trip from Heemskerk (Hk, see fig. 14.2) to Amsterdam CS (Asd) two routes appear to be relevant: route 1 (26 tariff kilometers) via Uitgeest (Utg), Zaandam (Zd) and Amsterdam Sloterdijk (Ass) and route 2 (35 km) via Haarlem (Hlm) and Ass. For a trip departing at 8:00 the following trip possibilities are suggested:

	Departure	Arrival	Time	Dist.	Changes	Route
(1)	7:27	8:05	0:38	26	Utg	(1)
(2)	7:32	8:11	0:39	35	-	(2)
(3)	7:57	8:35	0:38	26	Utg	(1)
(4)	8:02	8:41	0:39	35	-	(2)
(5)	8:27	9:05	0:38	26	Utg	(1)
(6)	8:32	9:14	0:42	35	-	(2)

Solution (4) is the initial solution, the other solutions are suggested as alternate solutions.

If the conventional time-tables are used to find solutions, then for a solution using route (1), say solution (3), two tables must be used. Table 42 a (see fig. 14.3) is used to find train 4829 from Hk to Utg, departing at 7:57 and arriving at 8:02. Then table 40 b (see fig. 14.4) is used to find train 14718 from Utg to Asd, departing at 8:07 and arriving at 8:35. For a solution using route (2), say solution (4), only one table needs to be used. Table 42 b (see fig. 14.5) is used to find train 4816 from Hk to Asd, departing at 8:02 and arriving at 8:41.

14.3.2. Hoorn to Den Haag HS

For a trip from Hoorn to Den Haag HS, the estimated travel time determined in SRM is 62 minutes. The upper bound is determined to be 87 minutes: the estimated travel time of 62 minutes plus 40 percent. The search space consists of 33 node stations, 14 percent of the entire network. The search space is given in fig. 14.6, including the relevant offset stations.

For a trip from Hoorn (Hn, see fig. 14.6) to Den Haag HS (Gv) two routes appear to be relevant: route 1 (97 km) via Purmerend (Pmr), Zaandam (Zd), Amsterdam Sloterdijk (Ass), Haarlem (Hlm), Leiden (Ledn) and route 2 (105 km) via Pmr, Zd, Ass, Amsterdam CS (Asd), Schiphol (Shl) and Ledn (the detour from

Ass to Asd is necessary since the intercity train from Asd to Gv via Shl does not stop at Ass). For a trip departing at 9:00 the following trip possibilities are suggested:

	Departure	Arrival	Time	Dist.	Changes	Route
(1)	8:15	9:41	1:26	105	Asd	(2)
(2)	8:37	9:56	1:19	97	Ass	(1)
(3)	8:45	10:11	1:26	105	Asd	(2)
(4)	9:07	10:26	1:19	97	Ass	(1)
(5)	9:37	10:56	1:19	97	Ass	(1)

Solution (4) is the initial solution, the other solutions are suggested as alternate solutions. Note the absence of a trip possibility at 9:15 and 9:45. The solutions at :15 and :45 are only possible in the early rush hours due to extra trains.

If the conventional time-tables are used to find solutions, then one is easily deceived by the layout of the time-tables. Amongst others, for the relation Hn - Gv connections via Alkmaar (Amr, see fig. 14.6), Beverwijk (Bv), Hlm and Ledn, are listed in table 41 b. However, *all* the listed connections from Hn to Gv are suboptimal! Let us consider the case of a departure at 9:00. Then table 41 b (see fig. 14.7) lists a connection departing at 9:05 (train 5539) and arriving at 10:41, changing to train 2139 at Ledn. By using two different tables, however, it is possible to find solution (4), which departs 2 minutes later and arrives 15 minutes earlier, also with one train change! For finding this solution, using route (1), table 40 b (see fig. 14.8) is used to find train 4522 from Hn to Ass, departing at 9:07 and arriving at 9:38. Note that since Hn is listed *three* times in table 40 b, and the lowest entry needs to be used, it is not easily found. Especially since the 9:05 train (5539) is also listed, but at the top. Then table 10 a (see fig. 14.9) is used to find train 5439 from Ass to Gv, departing at 9:45 and arriving at 10:26. For a solution using route (2), say solution (3), table 40 b (see fig. 14.10) is used to find train 14522 from Hn to Asd, departing at 8:45 and arriving at 9:18. Then table 10 a (see fig. 14.9) is used to find train 159 from Asd to Gv, departing at 9:26 and arriving at 10:11.

14.3.3. Den Haag CS to Blerick

For a trip from Den Haag CS to Blerick, the estimated travel time determined in SRM is 110 minutes. The upper bound is determined to be 154 minutes: the estimated travel time of 110 minutes plus 40 percent. The search space consists of 101 node stations, 43 percent of the entire network. The search space is given in fig. 14.11, including the relevant offset stations.

For a trip from Den Haag CS (Gvc, see fig. 14.11) to Blerick (Br) three routes appear to be relevant: route 1 (190 km) via Zoetermeer (Ztm), Gouda (Gd),

Utrecht CS (Ut), 's-Hertogenbosch (Ht), Eindhoven (Ehv), route 2 (181 km) via Rotterdam CS (Rtd), Dordrecht (Ddr), Breda (Bd), Ehv and route 3 (196 km) via Ztm, Gd, Ut, Arnhem (Ah), Nijmegen (Nm). For a trip departing at 9:00 the following trip possibilities are suggested:

	Departure	Arrival	Time	Dist.	Changes	Route
(1)	7:59	10:38	2:39	181	-	(2)
(2)	8:19	10:38	2:19	190	Ut, Ehv	(1)
(3)	8:35	11:16	2:41	196	Ut, Ah	(3)
(4)	8:59	11:38	2:39	181	-	(2)
(5)	9:19	11:38	2:19	190	Ut, Ehv	(1)
(6)	9:35	12:16	2:41	196	Ut, Ah	(3)

Solution (5) is the initial solution, the other solutions are suggested as alternate solutions. Note that solution (1) and (4) are suboptimal alternate solutions with fewer train changes to solutions (2) and (5) respectively.

If the conventional time-tables are used to find solutions, then for a solution using route (1), say solution (5), three tables must be used. Table 30 a (see fig. 14.12) is used to find train 2833 from Gvc to Ut, departing at 9:19 and arriving at 10:01. Then table 20 a (see fig. 14.13) is used to find train 833 from Ut to Ehv, departing at 10:04 and arriving at 10:57. Table 50 a (see fig. 14.14) is used to find train 1933 from Ehv to Br, departing at 10:59 and arriving at 11:38. For a solution using route (2), say solution (4), only one table needs to be used. Table 50 a (see fig. 14.14) is used to find train 1933 directly from Gvc to Br, departing at 8:59 and arriving at 11:38. Note that this train is spread across two columns without changing trains! For a solution using route (3), say solution (3), table 30 a (see fig. 14.15) is used to find train 529 from Gvc to Ut, departing at 8:35 and arriving at 9:17, and to find the connecting train 2929 from Ut to Ah, departing at 9:20 and arriving at 9:55. Table 51 a (see fig. 14.16) is used to find train 6235 from Ah to Br, departing at 10:06 and arriving at 11:16 (the 11:17 listed in this table is in fact the departure time of this train from Br; the TRAINS system uses the NS database which contains both the time of arrival and the time of departure at a station).

14.3.4. Vlissingen to Zwolle

For a trip from Vlissingen to Zwolle, the estimated travel time determined in SRM is 170 minutes. The upper bound is determined to be 230 minutes: the estimated travel time of 170 minutes plus the maximum detour of 60 minutes. The search space consists of 109 node stations, 46 percent of the entire network. The search space is given in fig. 14.17, including the relevant offset stations.

For a trip from Vlissingen (Vs, see fig. 14.17) to Zwolle (Zl) two routes appear to be relevant: route 1 (276 km) via Roosendaal (Rsd), Rotterdam CS (Rtd), Gouda (Gd), Utrecht CS (Ut) Amersfoort (Amf) and route 2 (280 km) via Rsd, Breda (Bd), 's-Hertogenbosch (Ht), Nijmegen (Nm), Arnhem (Ah), Zutphen (Zp). For a trip departing at 12:00 the following trip possibilities are suggested:

	Departure	Arrival	Time	Dist.	Changes	Route
(1)	11:20	14:46	3:26	276	Rsd, Rtd, Amf	(1)
(2)	11:20	15:10	3:50	280	Rsd	(2)
(3)	11:56	15:14	3:18	276	Rtd	(1)
(4)	12:20	15:46	3:26	276	Rsd, Rtd, Amf	(1)
(5)	12:20	16:10	3:50	280	Rsd	(2)
(6)	12:56	16:14	3:18	276	Rtd	(1)

Solution (4) is the initial solution, the other solutions are suggested as alternate solutions. Note that solution (2) and (5) are suboptimal alternate solutions with fewer train changes to solutions (1) and (4) respectively.

If the conventional time-tables are used to find solutions, then again the layout may be somewhat deceiving. For the relation Vs - Zl connections via route (2) are listed in table 60 b (see fig. 14.18). However, only some of the listed connections from Vs to Zl are suboptimal solutions with fewer train changes at best! Let us consider the case of a departure at 11:50. Then table 60 b (see fig. 14.18) lists a connection departing at 11:56 (train 2140) and arriving at 15:43, changing to train 4646 at Rsd. By using two different tables, however, it is possible to find solution (3), which departs at the same time (using in fact, the same train from Vs), but which arrives 29 minutes earlier, also with one train change! For finding this solution, using route (1), table 10 b (see fig. 14.19) is used to find train 2140 from Vs to Rtd, departing at 11:56 and arriving at 13:32. Then table 80 a (see fig. 14.20) is used to find train 549 from Rtd to Zl, departing at 13:39 and arriving at 15:14. For a solution using route (2), say solution (5), table 60 b (see fig. 14.18) is used to find train 14648 from Vs to Rsd, departing at 12:20 and arriving at 13:19. Then the same table is used to find train 4648 from Rsd to Zl, departing at 13:25 and arriving at 16:10. Note that solution (6) departs 36 minutes later while it arrives only 4 minutes later. Furthermore solution (4) departs at the same time, but arrives 24 minutes earlier, requiring 2 more train changes (this solution can be found by using table 10 b for the part Vs - Rsd - Rtd and table 80 a for the part Rtd - Amf - Zl).

14.4. Performance

In this section we shall look at the performance aspects of the algorithms used in TRAINS. First we shall look at the computational effects of the different techniques used, and then at the time requirements. The examples from the previous sections will be used.

14.4.1. Computational effects

In order to measure the effects of SRM and DYNET* we shall look at the amount of computation it took to find the initial solution of the four examples from the previous sections. The amount of computation is measured by the number of paths which became branching paths in step (3) of both the forward and the backward pass of the algorithm to search discrete dynamic networks (see section 6.6), and by the number of partial paths which were put in the frontier F in step (5). For each solution, we measured the computational effort, both with and without the SRM and DYNET* techniques.

First we consider the amount of computational effort necessary to find the fastest (initial) solution, without searching for an alternate suboptimal solution with fewer train changes. For the example Heemskerk to Amsterdam CS the results are:

Techniques used	branch paths fwd	branch paths bkwd	paths in frontier
DYNET	6	5	23
SRM + DYNET	6	5	21
DYNET*	4	5	20
SRM + DYNET*	4	5	18

For the example Hoorn to Den Haag HS the results are:

Techniques used	branch paths fwd	branch paths bkwd	paths in frontier
DYNET	33	6	79
SRM + DYNET	22	6	39
DYNET*	11	6	44
SRM + DYNET*	11	6	39

For the example Den Haag CS to Blerick the results are:

Techniques used	branch paths fwd	branch paths bkwd	paths in frontier
DYNET	179	12	261
SRM + DYNET	129	12	168
DYNET*	61	9	153
SRM + DYNET*	61	9	136

For the example Vlissingen to Zwolle the results are:

Techniques used	branch paths fwd	branch paths bkwd	paths in frontier
DYNET	157	11	227
SRM + DYNET	129	11	176
DYNET*	27	11	105
SRM + DYNET*	27	11	103

We now consider the amount of computational effort necessary to find not only the initial solution, but also (if possible) an alternate suboptimal solution with fewer train changes (using the techniques described in chapter 11). For the example Heemskerk to Amsterdam CS the results are:

Techniques used	branch paths fwd	branch paths bkwd	paths in frontier
DYNET	5	7	42
SRM + DYNET	5	7	35
DYNET*	4	6	40
SRM + DYNET*	4	6	33

For the example Hoorn to Den Haag HS the results are:

Techniques used	branch paths fwd	branch paths bkwd	paths in frontier
DYNET	36	7	149
SRM + DYNET	26	7	68
DYNET*	19	7	97
SRM + DYNET*	19	7	88

For the example Den Haag CS to Blerick the results are:

Techniques used	branch paths fwd	branch paths bkwd	paths in frontier
DYNET	303	14	484
SRM + DYNET	208	14	286
DYNET*	129	10	371
SRM + DYNET*	128	10	325

For the example Vlissingen to Zwolle the results are:

Techniques used	branch paths fwd	branch paths bkwd	paths in frontier
DYNET	216	46	411
SRM + DYNET	192	46	326
DYNET*	49	32	281
SRM + DYNET*	49	32	269

14.4.1.1. DYNET* versus SRM

From these figures it can be seen that the DYNET* extension is most important for reducing the search necessary to find solutions. SRM is less effective than DYNET* in reducing the number of branching paths. When DYNET* is used, SRM does not much reduce the number of branching paths further. This is not surprising since both SRM and DYNET* use the same information about estimates of remaining travel time, and since the upper limit estimate used in SRM is much coarser than the combination of actual time and estimates in DYNET*, which gradually gains more actual time information during search, and has to rely on estimates less and less. However, SRM does reduce the number of paths that are developed and put in the frontier, but which are never made a branching path. SRM prevents the development of paths which DYNET* would develop, put in the frontier, and successively deem unpromising. Furthermore, because of the (guided) depth first nature of DYNET*, relabeling occurs more often. In our implementation, relabeling means that a new path has to be added to the frontier. Therefore, DYNET* may require more paths to be put in the frontier. This is especially clear in the example Hoorn to Den Haag HS.

14.4.1.2. Limiting the backward search

It can be seen from the results that the techniques used to limit the (second) backward search by using information from the (first) forward search (see chapter 4), are very effective in case of a more complex question. With less complex problems, SRM and DYNET* do not further improve the second pass much. The accurate information from the first pass replaces the much coarser estimates used in SRM and DYNET*.

14.4.2. Time requirements

We now look at the time requirements and savings of the different techniques. For each example problem we have measured the amount of time that is required to compute the SRM search space and the amount of time that is required to compute estimates for DYNET* for the entire network. If SRM is used, for all vertices in the search space the estimates for DYNET* are generated as a side effect and the DYNET* computation does not need to be performed. Furthermore we have measured the time that is required to find the initial solution and the time to find all solutions necessary to completely answer a user's question. Note that more solutions may be found than those which are actually suggested to the user. For each example the required time is measured both with and without making use of the SRM and

DYNET* techniques. Only the pure searching time is measured, not including the time required for input and output. The measurements were performed on an Atari ST computer with an 8 Mhz Motorola MC 68000 micro processor, using a 200 Hz system timer.

First we consider the time required to find only the fastest solutions, without also searching for suboptimal solutions with fewer train changes.

For the example Heemskerk to Amsterdam CS it took 0.19 seconds to compute the DYNET* estimates for the entire network, and 0.05 seconds to compute the SRM search space. Note that, since SRM also determines the DYNET* estimates for the vertices in the search space, if SRM is applied, the DYNET* estimates do not have to be computed for the entire network. The other results are (in seconds):

Techniques used	first solution	all solutions
DYNET	0.10	2.52
SRM + DYNET	0.09	1.00
DYNET*	0.08	1.00
SRM + DYNET*	0.08	0.85

For the example Hoorn to Den Haag HS it took 0.19 seconds to compute the DYNET* estimates and 0.10 seconds to compute the SRM search space (including the DYNET* estimates for the vertices in the search space). The other results are (in seconds):

Techniques used	first solution	all solutions
DYNET	0.41	5.30
SRM + DYNET	0.28	2.15
DYNET*	0.24	1.82
SRM + DYNET*	0.22	1.65

For the example Den Haag CS to Blerick it took 0.19 seconds to compute the DYNET* estimates and 0.17 seconds to compute the SRM search space (including the DYNET* estimates for the vertices in the search space). The other results are (in seconds):

Techniques used	first solution	all solutions
DYNET	1.49	6.51
SRM + DYNET	1.04	4.85
DYNET*	0.68	2.77
SRM + DYNET*	0.62	2.63

For the example Vlissingen to Zwolle it took 0.19 seconds to compute the DYNET* estimates and 0.17 seconds to compute the SRM search space (including the DYNET* estimates for the vertices in the search space). The other results are (in seconds):

Techniques used	first solution	all solutions
DYNET	1.36	9.00
SRM + DYNET	1.15	6.85
DYNET*	0.49	3.17
SRM + DYNET*	0.48	3.06

We now consider the time required to find not only the fastest solutions, but also (if possible) suboptimal solutions with fewer train changes (using the techniques described in chapter 11). For the example Heemskerk to Amsterdam CS the results are (in seconds):

Techniques used	first solution	all solutions
DYNET	0.17	6.21
SRM + DYNET	0.15	1.79
DYNET*	0.16	2.13
SRM + DYNET*	0.15	1.68

For the example Hoorn to Den Haag HS the results are (in seconds):

Techniques used	first solution	all solutions
DYNET	1.14	13.35
SRM + DYNET	0.77	5.72
DYNET*	0.87	5.19
SRM + DYNET*	0.83	4.49

For the example Den Haag CS to Blerick the results are (in seconds):

Techniques used	first solution	all solutions
DYNET	6.64	23.56
SRM + DYNET	4.08	16.99
DYNET*	3.17	10.70
SRM + DYNET*	2.99	10.25

For the example Vlissingen to Zwolle the results are (in seconds):

Techniques used	first solution	all solutions
DYNET	4.79	27.80
SRM + DYNET	4.14	22.05
DYNET*	1.91	9.75
SRM + DYNET*	1.85	9.40

From the measurements it can be seen again that the DYNET* extension using the fastest train estimates is most effective in reducing running times. With less complex problems, applying SRM with the determination of both the search space and the DYNET* estimates for the vertices in the search space, is more efficient than the determination of the DYNET* estimates for the entire network only. In fact, determining the SRM search space with DYNET* estimates is never less efficient than determining the DYNET* estimates for the entire network. When only one solution is required, in case of a medium to complex problem the investment of applying SRM is returned when searching. When multiple solutions are required, applying SRM is always more efficient than not applying SRM. When DYNET* is used in combination with SRM the running times can be as much as three times faster. Note that in our implementation of SRM, we do not cut off the dead-end branches in the search space. If these branches would be cut off, then the performance of SRM might be improved.

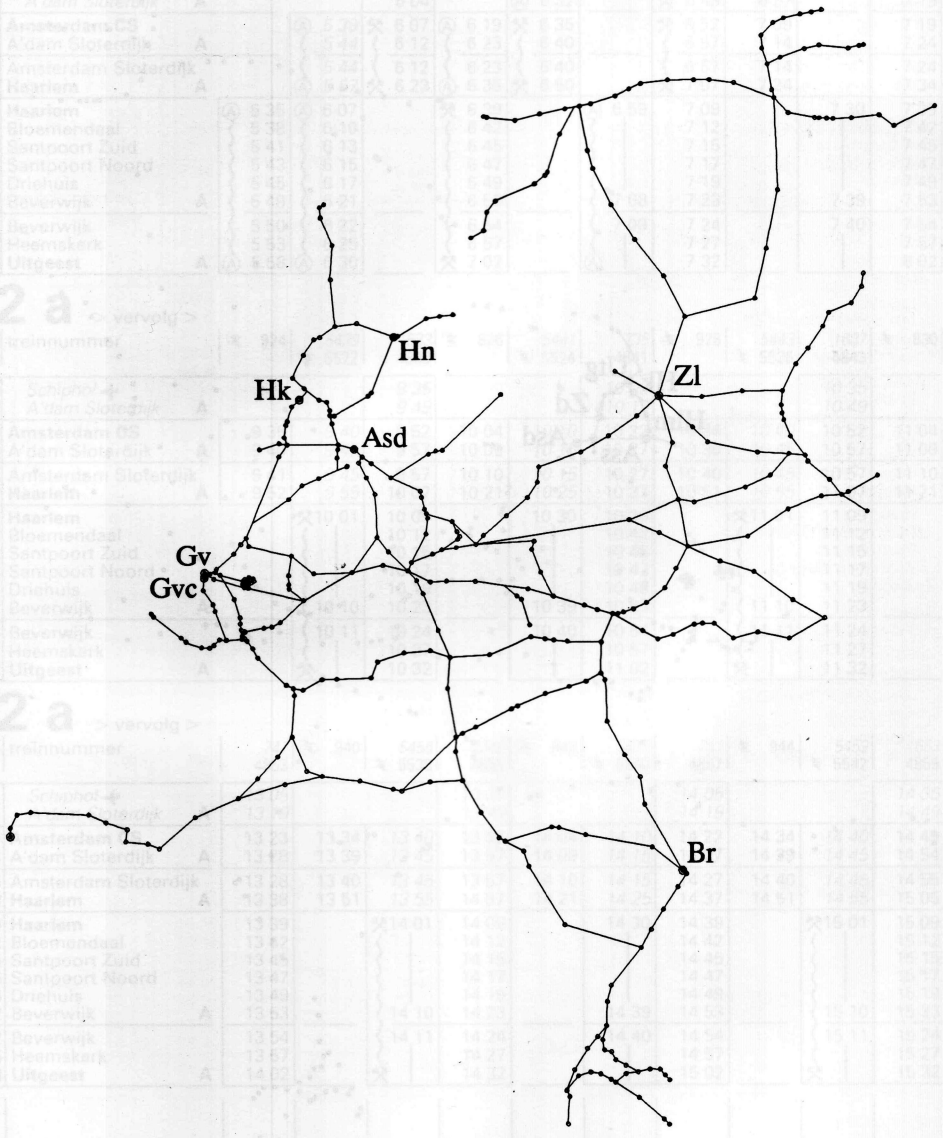


Fig. 14.1. The Dutch railway service network

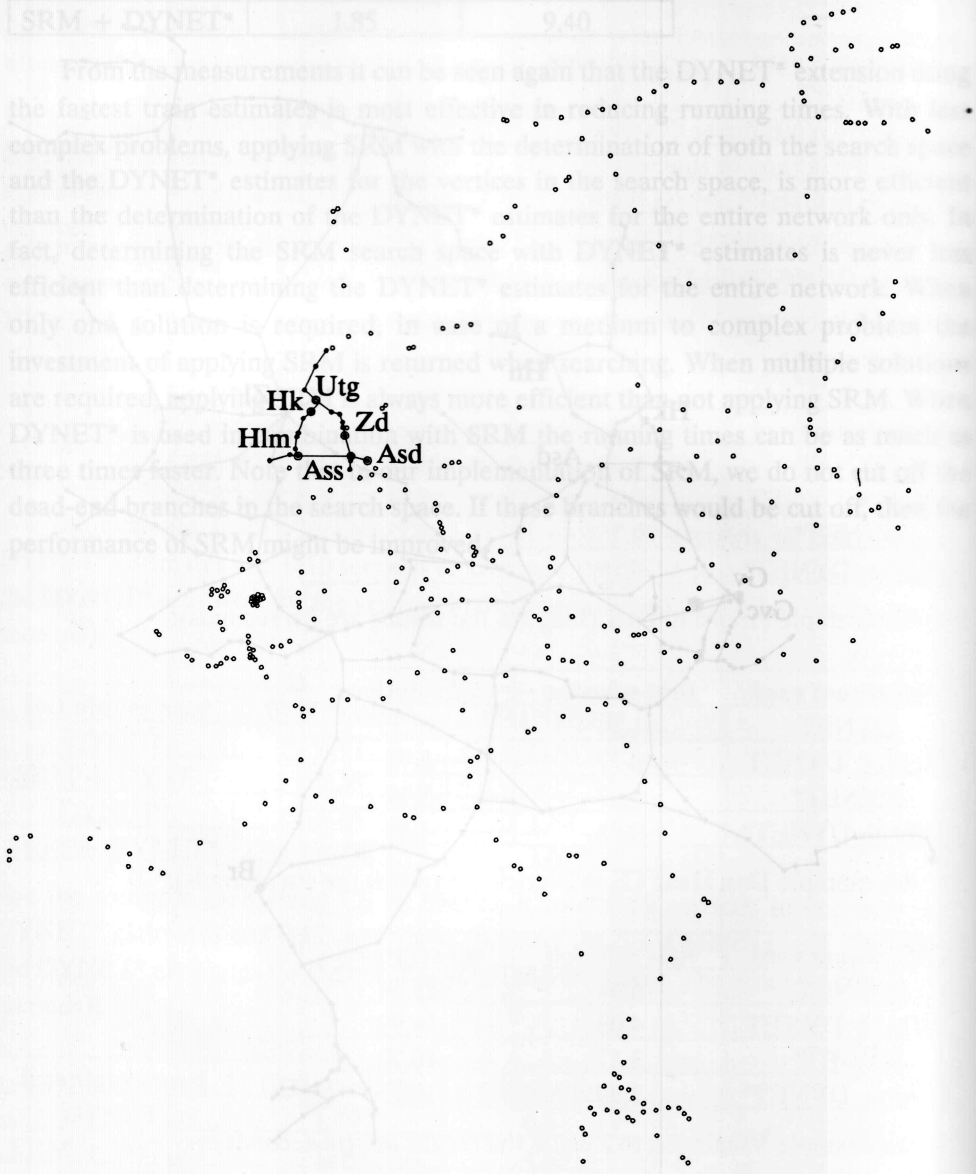


Fig. 14.2. The search space for Hk - Asd

42 a

km	treinnummer		4821	5321 4823	5004 5425	4825	2108 5427	5510	2241 4827	176 5429	5512	723 4829
	Schiphol ✈				*5 52*		Ⓐ *6 20*		*6 35*	*6 46*		*7 05*
	A'dam Sloterdijk	A			*6 04*		Ⓐ *6 32*		*6 49*	*6 57*		*7 19*
0	**Amsterdam CS**			Ⓐ *5 39*	⚒ 6 07	Ⓐ 6 19	⚒ 6 35		⚒ 6 52	7 09		7 19
4	A'dam Sloterdijk	A		*5 44*	6 12	6 23	6 40		6 57	7 14		7 24
4	Amsterdam Sloterdijk			*5 44*	6 12	6 23	6 40		6 57	7 14		7 24
19	**Haarlem**	A		Ⓐ *5 57*	⚒ 6 23	Ⓐ 6 35	⚒ 6 50		⚒ 7 07	7 24		7 34
19	**Haarlem**		Ⓐ 5 35	Ⓐ 6 07		⚒ 6 39		Ⓐ 6 59	7 09		7 30	7 39
22	Bloemendaal		5 38	6 10		6 42			7 12			7 42
24	Santpoort Zuid		5 41	6 13		6 45			7 15			7 45
26	Santpoort Noord		5 43	6 15		6 47			7 17			7 47
26	Driehuis		5 45	6 17		6 49			7 19			7 49
32	Beverwijk	A	5 49	6 21		6 53		7 08	7 23		7 39	7 53
32	Beverwijk		5 50	6 22		6 54		7 09	7 24		7 40	7 54
35	Heemskerk		5 53	6 25		6 57			7 27			7 57
38	**Uitgeest**	A	Ⓐ 5 58	Ⓐ 6 30		⚒ 7 02		Ⓐ	7 32			8 02

42 a > vervolg >

km	treinnummer		924	5439 5522	1633 4839	826	5441 5524	735 4841	928	5443 5526	1637 4843	830
	Schiphol ✈				*9 35*			*10 05*			*10 35*	
	A'dam Sloterdijk	A			*9 49*			*10 19*			*10 49*	
0	**Amsterdam CS**		9 35	*9 40*	9 52	10 04	*10 10*	10 22	10 34	*10 40*	10 52	11 04
4	A'dam Sloterdijk	A	9 40	*9 45*	9 57	10 09	*10 15*	10 27	10 39	*10 45*	10 57	11 09
4	Amsterdam Sloterdijk		9 41	*9 45*	9 57	10 10	*10 15*	10 27	10 40	*10 45*	10 57	11 10
19	**Haarlem**	A	9 52	*9 55*	10 07	10 21	*10 25*	10 37	10 51	*10 55*	11 07	11 21
19	**Haarlem**			⚒ 10 01	10 09		10 30	10 39		⚒ 11 01	11 09	
22	Bloemendaal				10 12			10 42			11 12	
24	Santpoort Zuid				10 15			10 45			11 15	
26	Santpoort Noord				10 17			10 47			11 17	
26	Driehuis				10 19			10 49			11 19	
32	Beverwijk	A		10 10	10 23		10 39	10 53		11 10	11 23	
32	Beverwijk			10 11	10 24		10 40	10 54		11 11	11 24	
35	Heemskerk				10 27			10 57			11 27	
38	**Uitgeest**	A		⚒	10 32			11 02		⚒	11 32	

42 a > vervolg >

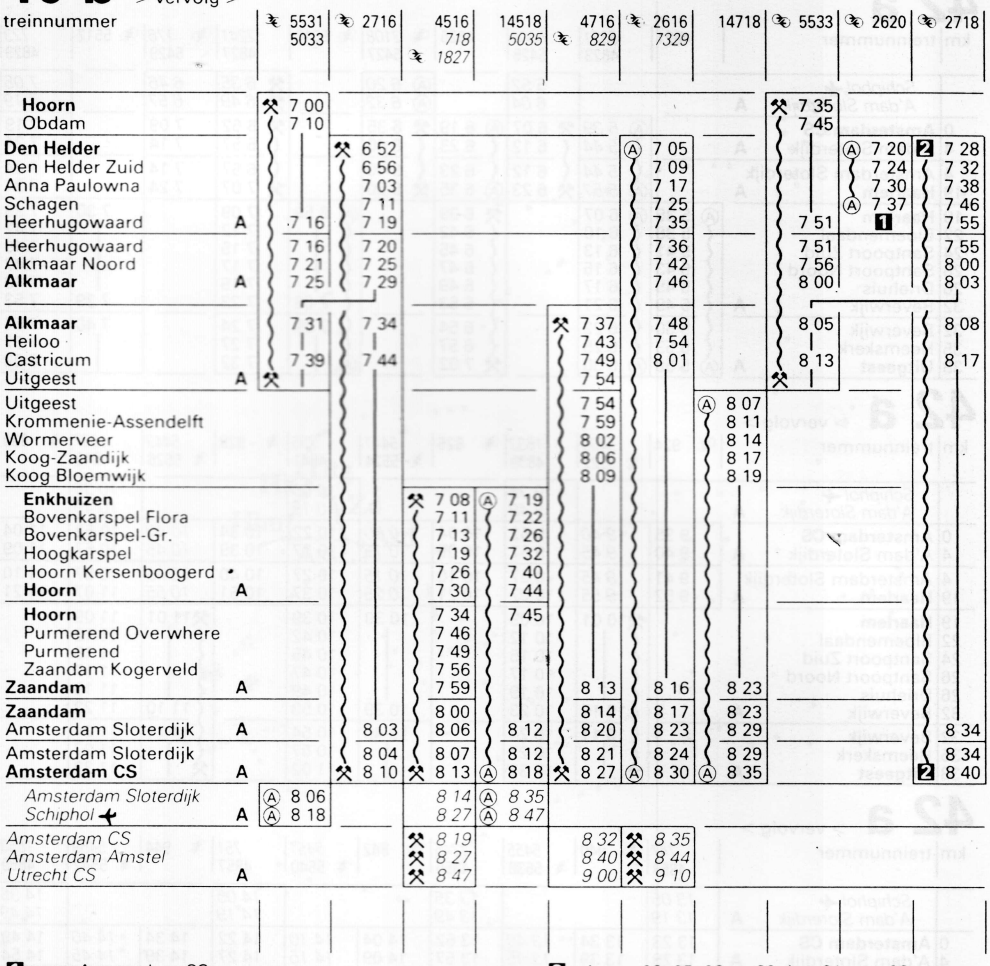
km	treinnummer		747 4853	940	5455 5538	2245 4855	842	5457 5540	751 4857	944	5459 5542	1653 4859
	Schiphol ✈		*13 05*			*13 35*			*14 05*			*14 35*
	A'dam Sloterdijk	A	*13 19*			*13 49*			*14 19*			*14 49*
0	**Amsterdam CS**		13 23	13 34	*13 40*	13 52	14 04	*14 10*	14 22	14 34	*14 40*	14 49
4	A'dam Sloterdijk	A	13 28	13 39	*13 45*	13 57	14 09	*14 15*	14 27	14 39	*14 45*	14 54
4	Amsterdam Sloterdijk		13 28	13 40	*13 45*	13 57	14 10	*14 15*	14 27	14 40	*14 45*	14 55
19	**Haarlem**	A	13 38	13 51	*13 55*	14 07	14 21	*14 25*	14 37	14 51	*14 55*	15 05
19	**Haarlem**		13 39		⚒ 14 01	14 09		14 30	14 39		⚒ 15 01	15 09
22	Bloemendaal		13 42			14 12			14 42			15 12
24	Santpoort Zuid		13 45			14 15			14 45			15 15
26	Santpoort Noord		13 47			14 17			14 47			15 17
26	Driehuis		13 49			14 19			14 49			15 19
32	Beverwijk	A	13 53		14 10	14 23		14 39	14 53		15 10	15 23
32	Beverwijk		13 54		14 11	14 24		14 40	14 54		15 11	15 24
35	Heemskerk		13 57			14 27			14 57			15 27
38	**Uitgeest**	A	14 02		⚒	14 32			15 02		⚒	15 32

1 **Boottrein London 🎩;** niet op 25 en 26 dec

2 niet op 23, 25, 26 en 30 dec, 31 mrt, 28 apr en 19 mei

Fig. 14.3.

40 b > vervolg >



1 naar Amsterdam CS
(zie enkele kolommen verder)

2 niet op 23, 25, 26 en 30 dec, 31 mrt, 28 apr en 19 mei

Fig. 14.4.

	5529	927	5012	4814	5531	5414	829	5014	4816	5533	5416	931	5016
	5412	2131	5033	718		5035			1620		5037		
Ui				7 29					7 59				
He				7 32					8 02				
Be	7 18			7 35	7 48				8 05	8 23			
Be	7 22			7 36	7 52				8 06	8 23			
Dr				7 40					8 10				
SN				7 42					8 12				
SZ				7 45					8 15				
Bl				7 48					8 18				
Ha	7 32			7 52	8 02				8 22	8 33			
Ha	*7 36*	7 41	Ⓐ 7 48	7 57		8 06	8 12	Ⓐ 8 18	8 25		8 37	8 42	Ⓐ 8 50
AS	*7 45*	7 51	7 58	8 06		8 15	8 22	8 27	8 34		8 46	8 52	9 00
AS	*7 45*	7 52	7 58	8 06		8 15	8 23	8 28	8 35		8 47	8 53	9 00
AC	*7 51*	7 58	Ⓐ 8 04	8 11		8 21	8 29	Ⓐ 8 34	8 41		8 53	8 59	Ⓐ 9 05
AS		*8 01*	Ⓐ *8 06*	*8 14*		Ⓐ *8 35*			*8 44*		**1** *8 51*		
Sc		*8 12*	Ⓐ *8 18*	*8 27*		Ⓐ *8 47*			*8 58*		**1** *9 03*		

	837	4824	5541	939	4826	5543	841	4828	5545	943	4830	5547	845
		1628	*5424*		*730*	*5426*		*2240*	*5428*		*734*	*5430*	
Ui		9 59			10 29			10 59			11 29		
He		10 02			10 32			11 02			11 32		
Be		10 05	10 22		10 35	10 52		11 05	11 22		11 35	11 52	
Be		10 06	10 22		10 36	10 52		11 06	11 22		11 36	11 52	
Dr		10 10			10 40			11 10			11 40		
SN		10 12			10 42			11 12			11 42		
SZ		10 15			10 45			11 15			11 45		
Bl		10 18			10 48			11 18			11 48		
Ha		10 22	10 32		10 52	11 02		11 22	11 32		11 52	12 02	
Ha	10 12	10 25	*10 36*	10 42	10 55	*11 06*	11 12	11 25	*11 36*	11 42	11 55	*12 06*	12 12
AS	10 22	10 34	*10 45*	10 52	11 04	*11 15*	11 22	11 34	*11 45*	11 52	12 04	*12 15*	12 22
AS	10 23	10 35	*10 45*	10 53	11 05	*11 15*	11 23	11 35	*11 45*	11 53	12 05	*12 15*	12 23
AC	10 29	10 41	*10 51*	10 59	11 11	*11 21*	11 29	11 41	*11 51*	11 59	12 11	*12 21*	12 29
AS		*10 44*			*11 14*			*11 44*			*12 14*		
Sc		*10 58*			*11 27*			*11 58*			*12 27*		

	5555	853	4840	5557	955	4842	5559	857	4844	5561	959	4846	5563
	5438		*1644*	*5440*		*746*	*5442*		*2244*	*5444*		*750*	*5446*
Ui			13 59			14 29			14 59			15 29	
He			14 02			14 32			15 02			15 32	
Be	13 52		14 05	14 22		14 35	14 52		15 05	15 22		15 35	15 52
Be	13 52		14 06	14 22		14 36	14 52		15 06	15 22		15 36	15 52
Dr			14 10			14 40			15 10			15 40	
SN			14 12			14 42			15 12			15 42	
SZ			14 15			14 45			15 15			15 45	
Bl			14 18			14 48			15 18			15 48	
Ha	14 02		14 22	14 32		14 52	15 02		15 22	15 32		15 52	16 02
Ha	*14 06*	14 12	14 25	*14 37*	14 42	14 55	*15 06*	15 12	15 25	*15 37*	15 42	15 55	*16 06*
AS	*14 15*	14 22	14 34	*14 46*	14 52	15 04	*15 15*	15 22	15 34	*15 46*	15 52	16 04	*16 15*
AS	*14 15*	14 23	14 35	*14 47*	14 53	15 05	*15 15*	15 23	15 35	*15 47*	15 53	16 05	*16 15*
AC	*14 21*	14 29	14 41	*14 53*	14 59	15 11	*15 21*	15 29	15 41	*15 53*	15 59	16 11	*16 21*
AS			*14 44*			*15 14*			*15 44*			*16 14*	
Sc			*14 58*			*15 27*			*15 58*			*16 27*	

> zie vervolg >

Fig. 14.5.

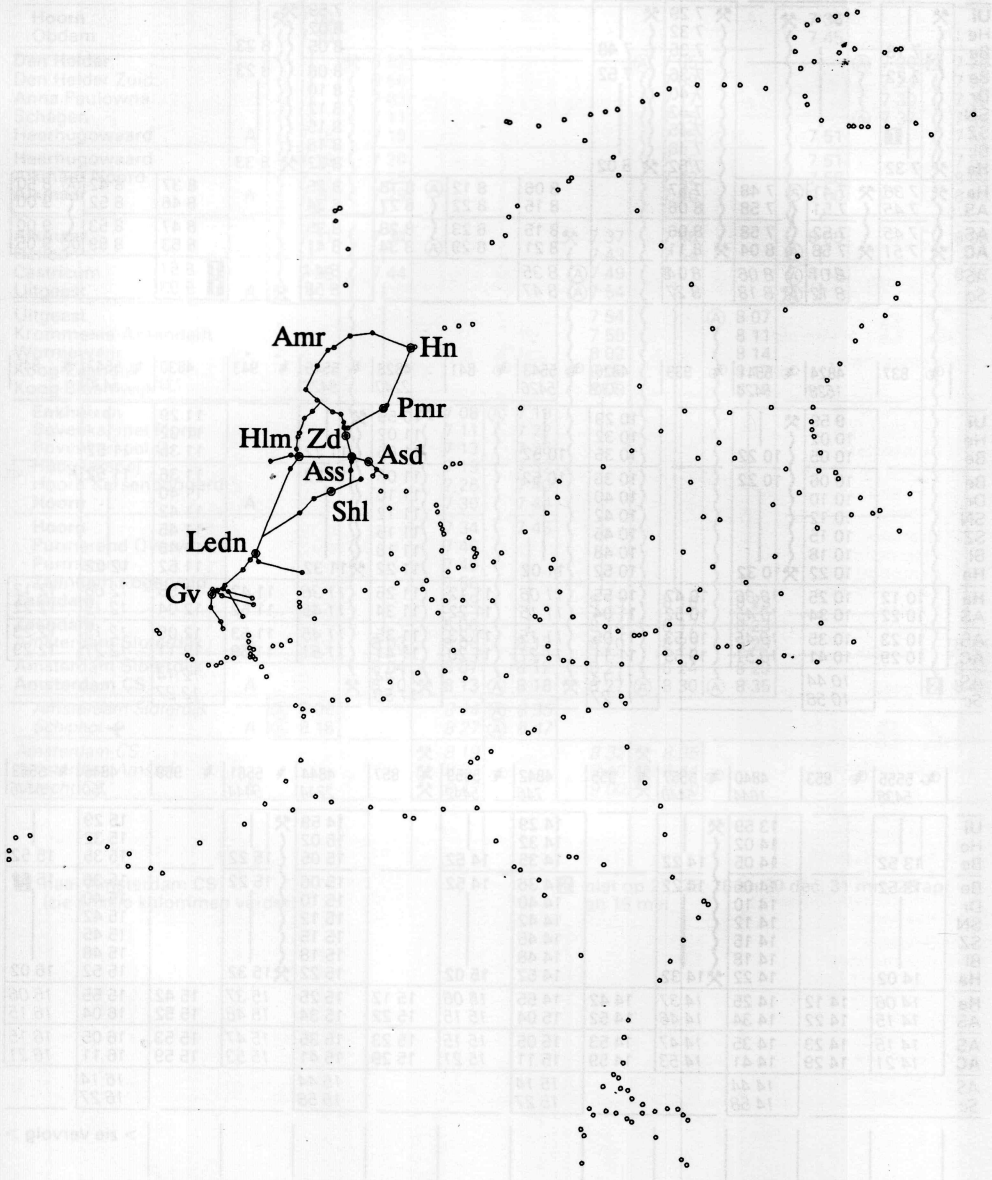
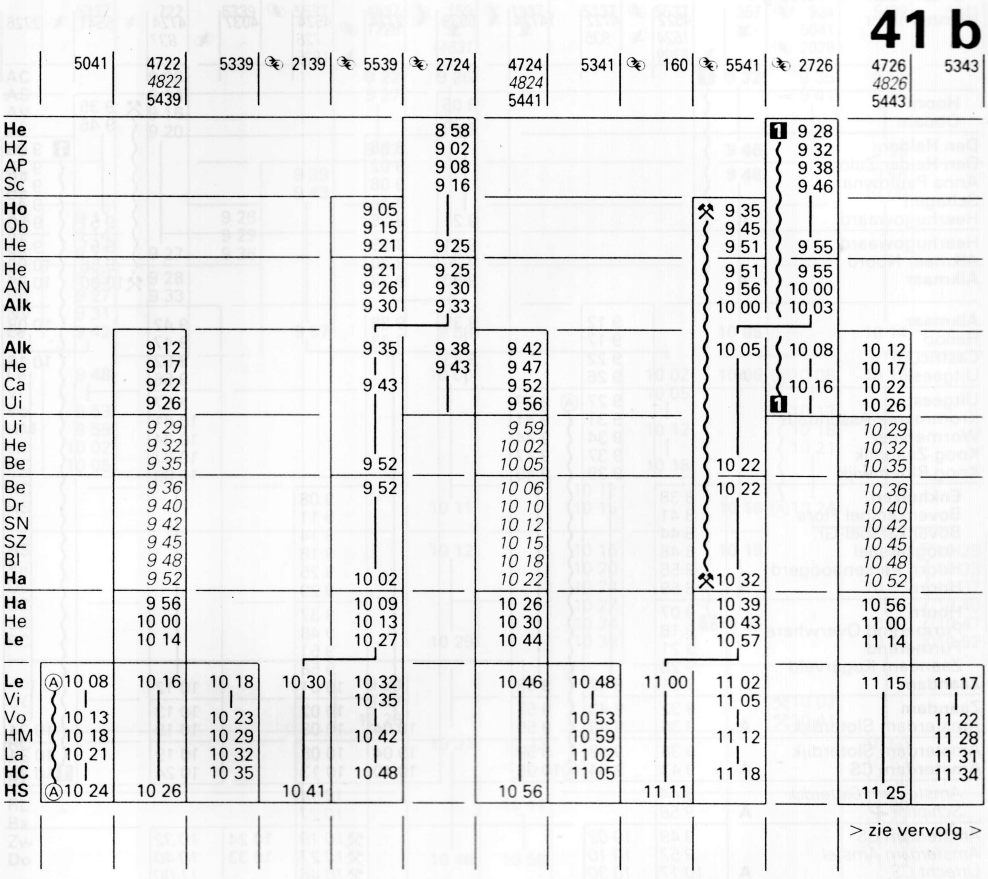


Fig. 14.6. The search space for Hn - Gv



41 b

	5041	4722 *4822* 5439	5339	2139	5539	2724	4724 *4824* 5441	5341	160	5541	2726	4726 *4826* 5443	5343
He						8 58					1 9 28		
HZ						9 02					9 32		
AP						9 08					9 38		
Sc						9 16					9 46		
Ho					9 05					9 35			
Ob					9 15					9 45			
He					9 21	9 25				9 51	9 55		
He					9 21	9 25				9 51	9 55		
AN					9 26	9 30				9 56	10 00		
Alk					9 30	9 33				10 00	10 03		
Alk		9 12			9 35	9 38	9 42			10 05	10 08	10 12	
He		9 17				9 43	9 47					10 17	
Ca		9 22			9 43		9 52				10 16	10 22	
Ui		9 26					9 56				1	10 26	
Ui		*9 29*					*9 59*					*10 29*	
He		*9 32*					*10 02*					*10 32*	
Be		*9 35*			9 52		*10 05*			10 22		*10 35*	
Be		*9 36*			9 52		*10 06*			10 22		*10 36*	
Dr		*9 40*					*10 10*					*10 40*	
SN		*9 42*					*10 12*					*10 42*	
SZ		*9 45*					*10 15*					*10 45*	
Bl		*9 48*					*10 18*					*10 48*	
Ha		*9 52*			10 02		*10 22*			10 32		*10 52*	
Ha		9 56			10 09		10 26			10 39		10 56	
He		10 00			10 13		10 30			10 43		11 00	
Le		10 14			10 27		10 44			10 57		11 14	
Le	Ⓐ 10 08	10 16	10 18	10 30	10 32		10 46	10 48	11 00	11 02		11 15	11 17
Vi					10 35					11 05			
Vo	10 13		10 23					10 53					11 22
HM	10 18		10 29		10 42			10 59		11 12			11 28
La	10 21		10 32					11 02					11 31
HC			10 35		10 48			11 05		11 18			11 34
HS	Ⓐ 10 24	10 26		10 41			10 56		11 11			11 25	

> zie vervolg >

Fig. 14.7.

40 b > vervolg >

treinnummer		4522 *1624* 2309	4722 *935*	14724	5539	2724	4524 *726* *1835*	4037	4724 *837*	5541	2726
Hoorn					9 05					⚒ 9 35	
Obdam					9 15					9 45	
Den Helder						8 58					1 9 28
Den Helder Zuid						9 02					9 32
Anna Paulowna						9 08					9 38
Schagen						9 16					9 46
Heerhugowaard	A				9 21	9 25				9 51	9 55
Heerhugowaard					9 21	9 25				9 51	9 55
Alkmaar Noord					9 26	9 30				9 56	10 00
Alkmaar	A				9 30	9 33				⚒ 10 00	10 03
Alkmaar			9 12		9 35	9 38			9 42		10 08
Heiloo			9 17			9 43			9 47		
Castricum			9 22		9 43				9 52		10 16
Uitgeest	A		9 26						9 56		
Uitgeest			9 27	Ⓐ 9 37					9 57		
Krommenie-Assendelft			9 31	9 41					10 01		
Wormerveer			9 34	9 44					10 04		
Koog-Zaandijk			9 37	9 47					10 07		
Koog Bloemwijk			9 39	9 49					10 09		
Enkhuizen		8 38					9 08				
Bovenkarspel Flora		8 41					9 11				
Bovenkarspel-Gr.		8 44					9 14				
Hoogkarspel		8 48					9 18				
Hoorn Kersenboogerd		8 55					9 25				
Hoorn	A	8 59					9 29				
Hoorn		9 07					9 37				
Purmerend Overwhere		9 18					9 48				
Purmerend		9 21					9 51				
Zaandam Kogerveld		9 28					9 58				
Zaandam	A	9 31	9 42	9 53			10 01		10 12		
Zaandam		9 32	9 43	9 53			10 02		10 13		
Amsterdam Sloterdijk	A	9 38	9 49	9 59		10 04	10 08		10 19		10 34
Amsterdam Sloterdijk		9 38	9 49	9 59		10 04	10 08		10 19		10 34
Amsterdam CS	A	9 43	9 54	Ⓐ 10 05		10 10	10 13		10 24		1 10 40
Amsterdam Sloterdijk		*9 44*					*10 14*				
Schiphol ✈	A	*9 58*					*10 27*				
Amsterdam CS		*9 49*	*10 02*				⚒ *10 19*	10 24	*10 32*		
Amsterdam Amstel		*9 57*	*10 10*				⚒ *10 27*	10 33	*10 40*		
Utrecht CS	A	*10 17*	*10 30*				⚒ *10 46*		*11 00*		

1 niet op 23, 25, 26 en 30 dec, 31 mrt, 28 apr en 19 mei

Fig. 14.8.

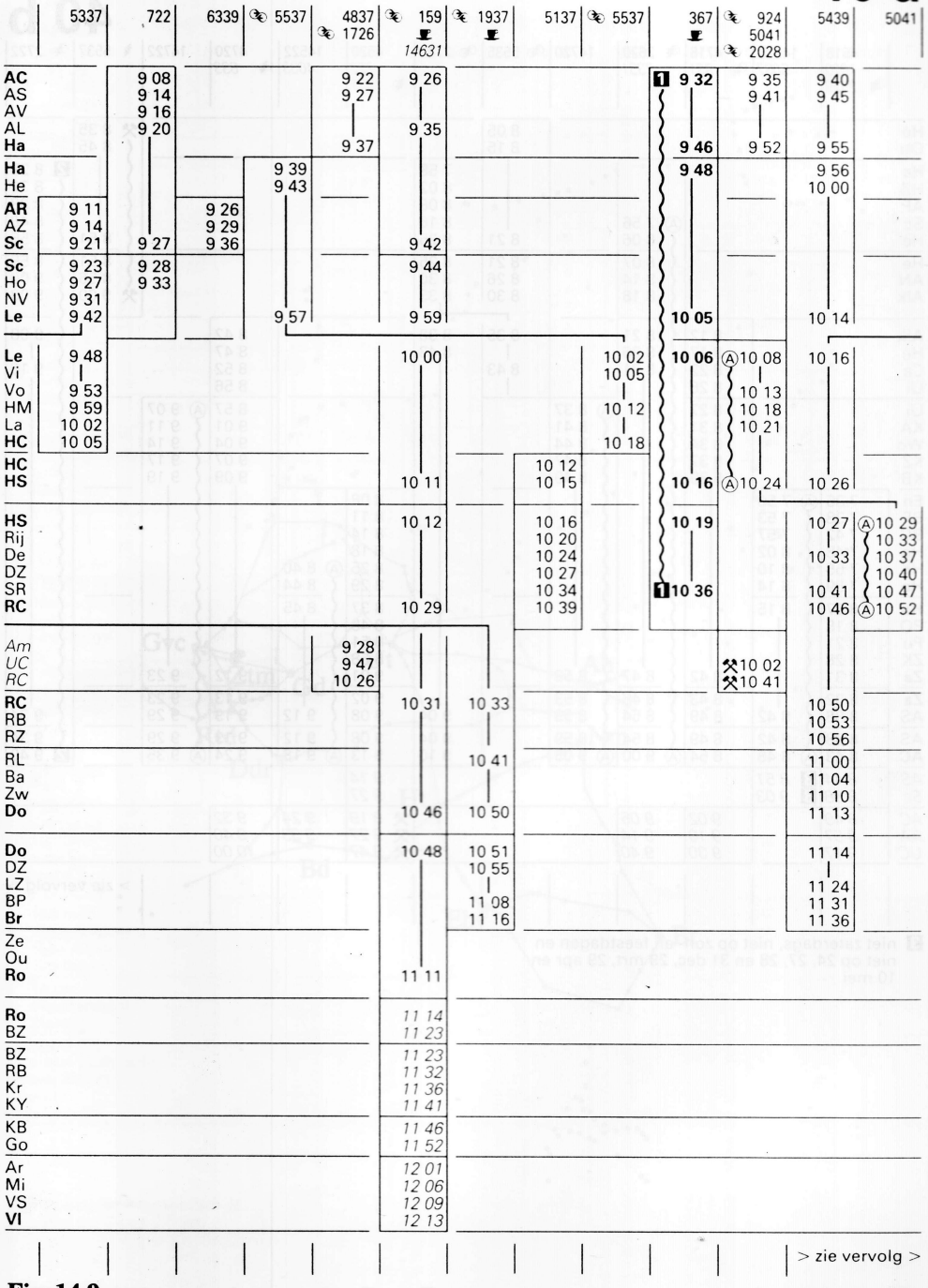
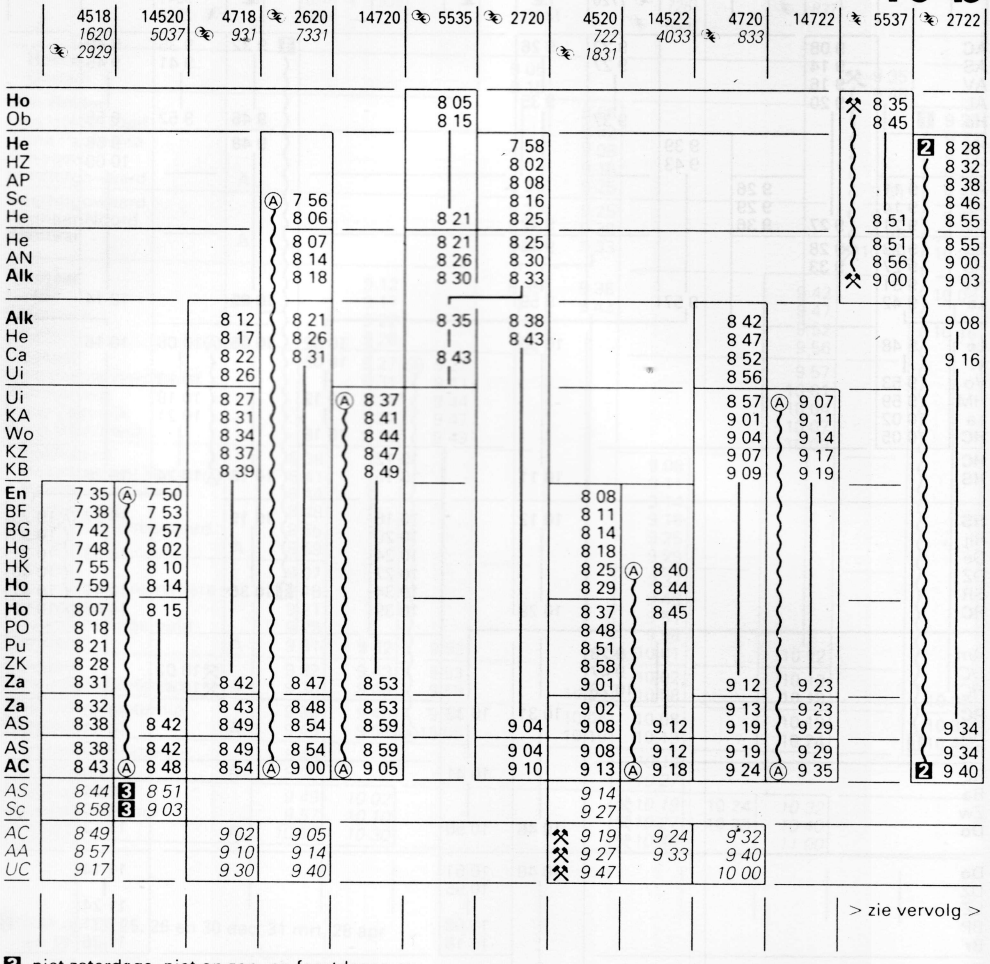

5337
722
6339
5537
4837
1726
159
14631
1937
5137
5537
367
924
5041
2028
5439
5041
AC 9 08 9 22 9 26 1 9 32 9 35 9 40
AS 9 14 9 27 9 41 9 45
AV 9 16
AL 9 20 9 35
Ha 9 37 9 46 9 52 9 55
Ha 9 39 9 48 9 56
He 9 43 10 00
AR 9 11 9 26
AZ 9 14 9 29
Sc 9 21 9 27 9 36 9 42
Sc 9 23 9 28 9 44
Ho 9 27 9 33
NV 9 31
Le 9 42 9 57 9 59 10 05 10 14
Le 9 48 10 00 10 02 10 06 Ⓐ10 08 10 16
Vi 10 05 10 13
Vo 9 53 10 12 10 18
HM 9 59
La 10 02
HC 10 05 10 12
HS 10 11 10 15 10 16 Ⓐ10 24 10 26
HS 10 12 10 16 10 19 10 27 Ⓐ10 29
Rij 10 20 10 33 10 33
De 10 24 10 37
DZ 10 27 10 40
SR 10 34 1 10 36 10 41 10 47
RC 10 29 10 39 10 46 Ⓐ10 52
Am 9 28
UC 9 47 ⚒10 02
RC 10 26 ⚒10 41
RC 10 31 10 33 10 50
RB 10 53
RZ 10 56
RL 10 41 11 00
Ba 11 04
Zw 11 10
Do 10 46 10 50 11 13
Do 10 48 10 51 11 14
DZ 10 55 11 24
LZ 11 08 11 31
BP 11 16 11 36
Br
Ze
Ou
Ro 11 11
Ro 11 14
BZ 11 23
BZ 11 23
RB 11 32
Kr 11 36
KY 11 41
KB 11 46
Go 11 52
Ar 12 01
Mi 12 06
VS 12 09
VI 12 13
> zie vervolg >

Fig. 14.9.



3 niet zaterdags, niet op zon- en feestdagen en niet op 24, 27, 28 en 31 dec, 29 mrt, 29 apr en 10 mei

Fig. 14.10.

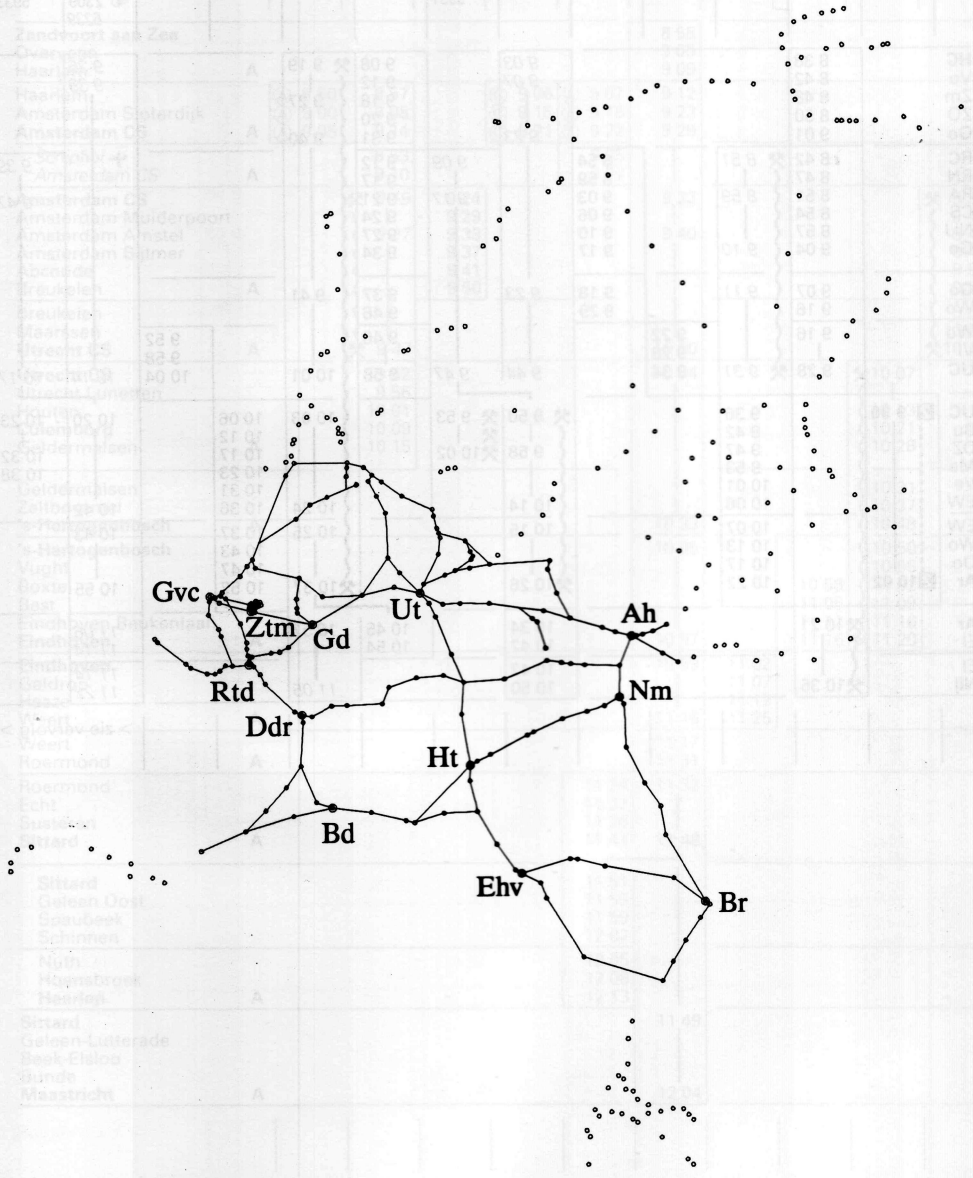


Fig. 14.11. The search space for Gvc - Br

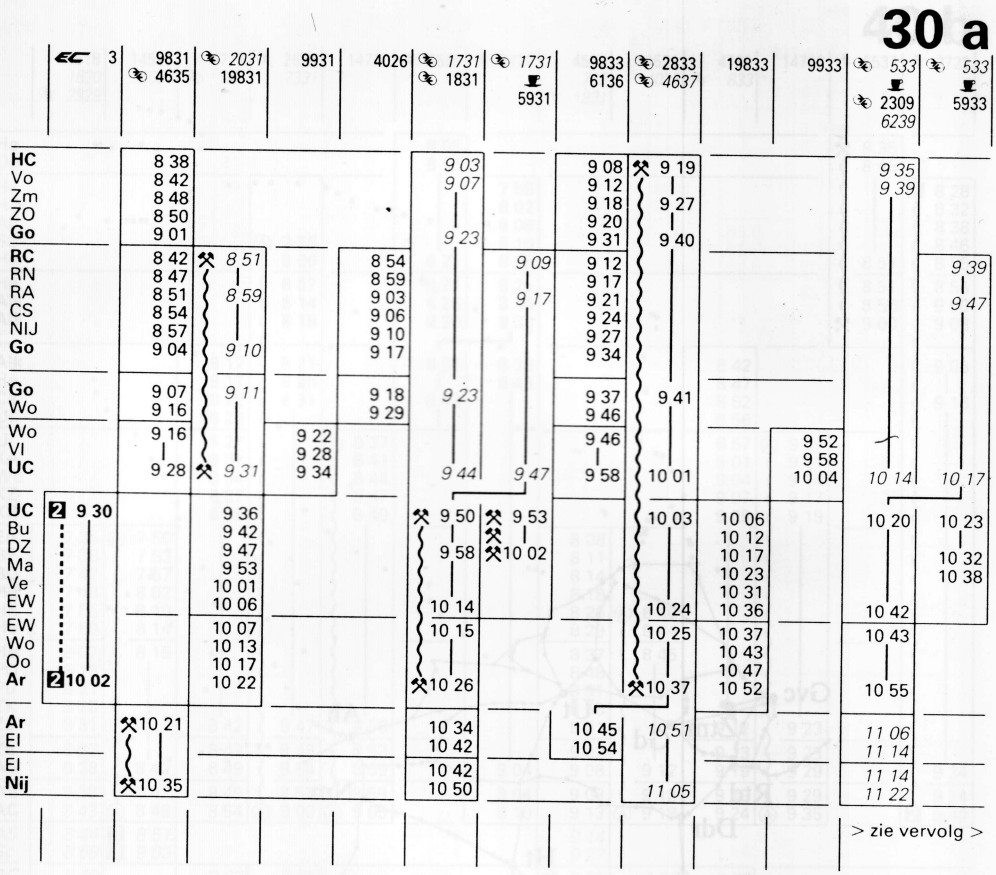
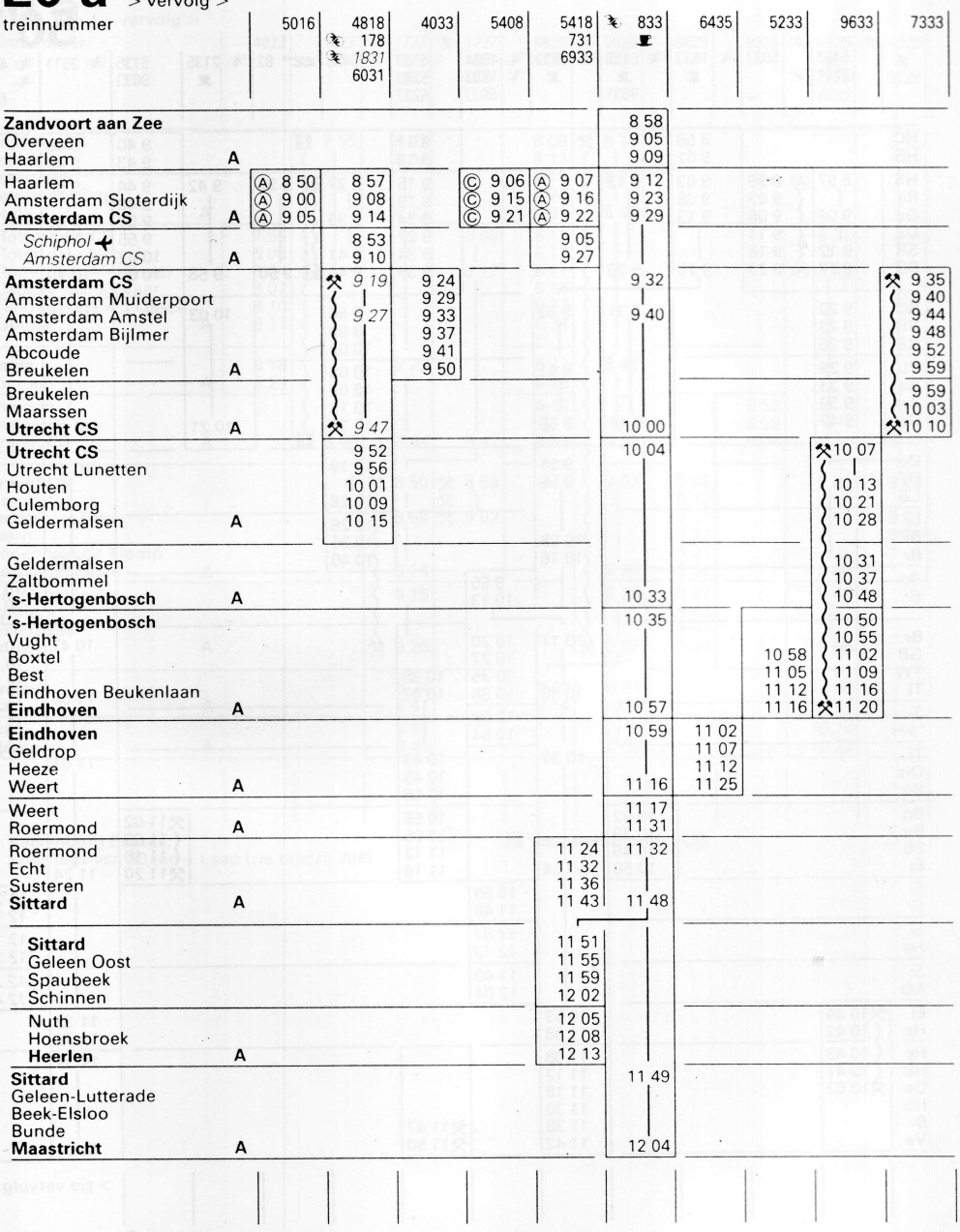

30 a
3
9831
4635
2031
19831
9931
4026
1731
1831
1731
5931
9833
6136
2833
4637
19833
9933
533
2309
6239
533
5933
HC
Vo
Zm
ZO
Go
RC
RN
RA
CS
NIJ
Go
Go
Wo
Wo
Vl
UC
UC
Bu
DZ
Ma
Ve
EW
EW
Wo
Oo
Ar
Ar
El
El
Nij
2 9 30
2 10 02
10 21
10 35
> zie vervolg >

Fig. 14.12.

20 a > vervolg >



1 **Boottrein Amsterdam**; niet op 26 en 27 dec

Fig. 14.13.

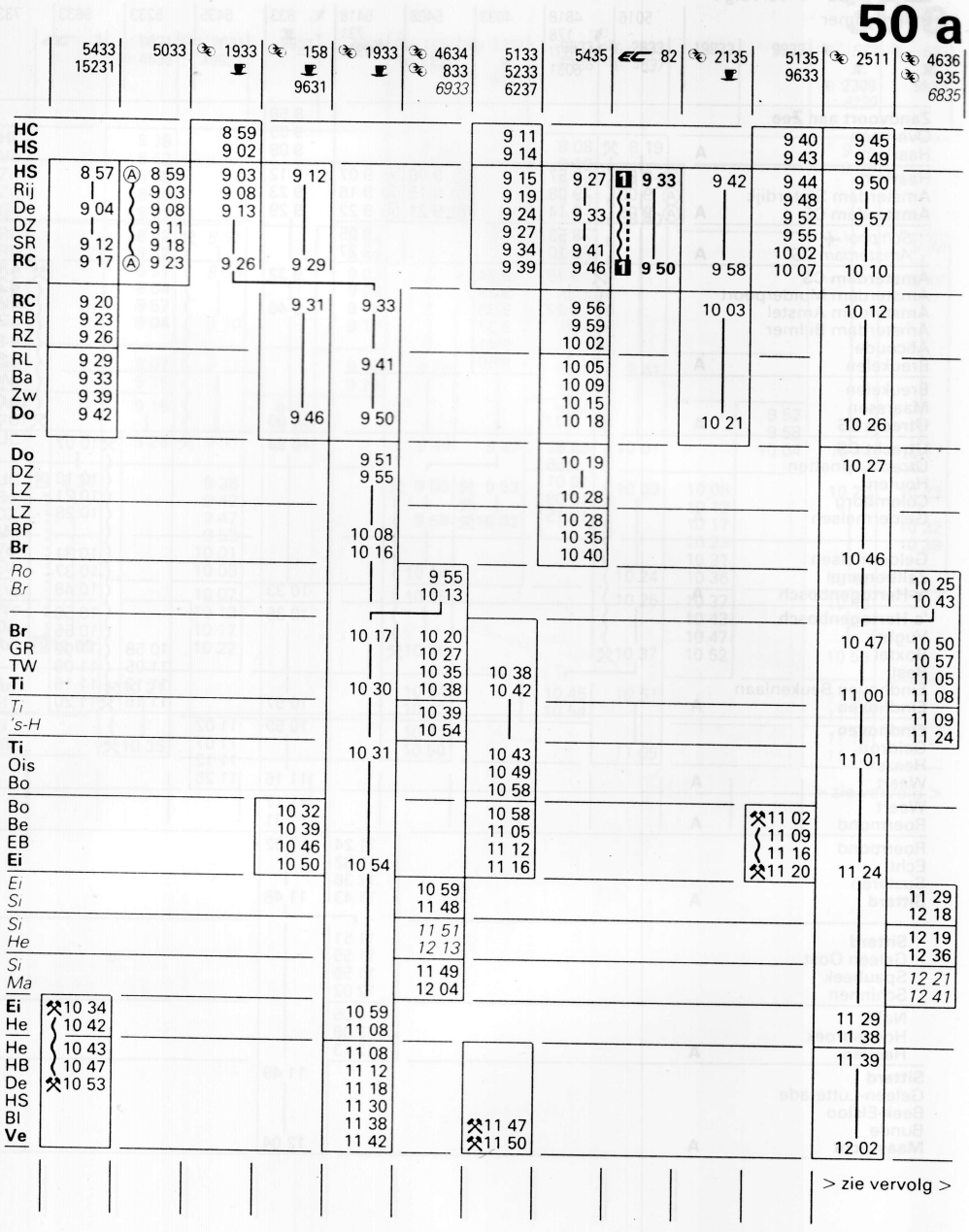
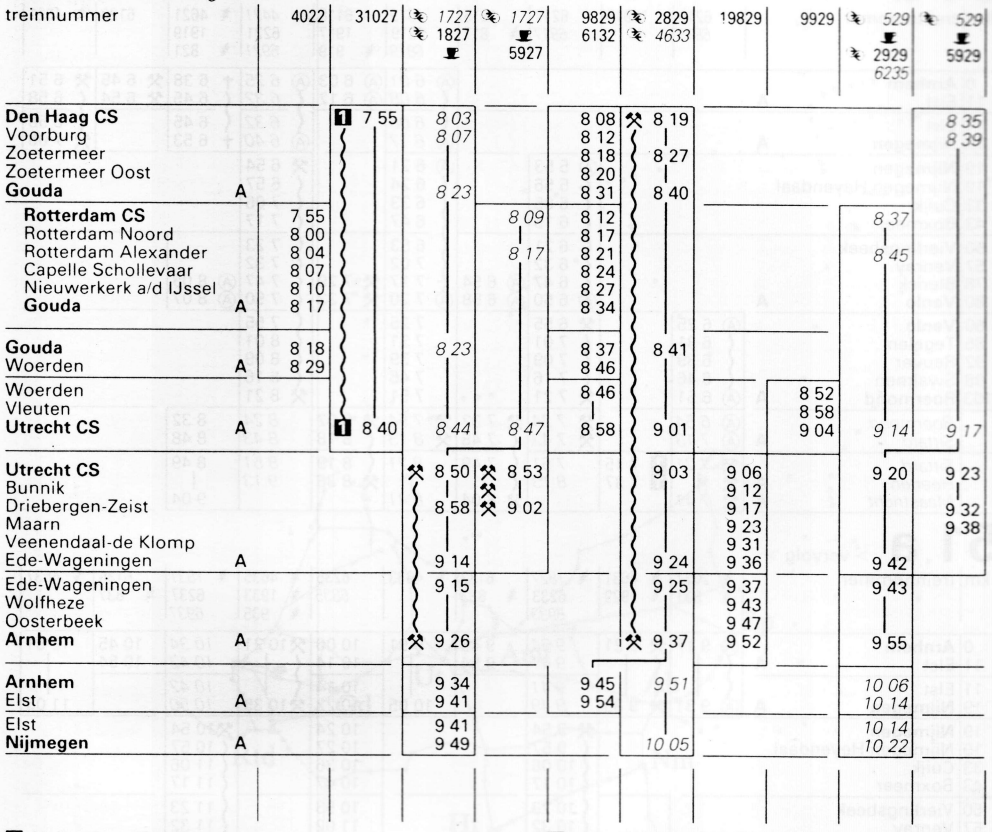

50 a
5433 15231
5033
1933
158 9631
1933
4634 833 6933
5133 5233 6237
5435
82
2135
5135 9633
2511
4636 935 6835
HC HS
HS Rij De DZ SR RC
RC RB RZ
RL Ba Zw Do
Do DZ LZ
LZ BP Br
Ro Br
Br GR TW Ti
Ti 's-H
Ti Ois Bo
Bo Be EB Ei
Ei Si
Si He
Si Ma
Ei He
He HB De HS Bl Ve
> zie vervolg >

Fig. 14.14.

30 a > vervolg >



1 **Valkenburg Expres;**
zaterdags van 30 jun—1 sep (zie bladzij 206)

2 **Rembrandt** R

Fig. 14.15.

51 a

km	treinnummer		6215 *6815*	16915	6217 *6917*	1915 817	*4619* 6219 *6819*	6118 1917 919	*4421* 6221 *6921*	4621 1919 821	6120	4621
0	**Arnhem**						Ⓐ *6 01*	Ⓐ 6 03	Ⓐ *6 25*	† 6 38	⚒ 6 45	⚒ 6 51
11	Elst	A					*6 08*	Ⓐ 6 12	*6 32*	6 45	⚒ 6 54	6 58
11	Elst						*6 08*		*6 32*	6 45		6 58
19	**Nijmegen**	A					Ⓐ *6 17*		Ⓐ *6 40*	† 6 53		⚒ 7 06
19	**Nijmegen**				Ⓐ 5 53		Ⓐ 6 21		⚒ 6 54			
19	Nijmegen Heyendaal				5 56		6 24		6 57			
33	Cuijk				6 05		6 33		7 06			
43	Boxmeer				6 15		6 47		7 17			
50	Vierlingsbeek				6 21		6 53		7 23			
57	Venray				6 32		7 02		7 32			
78	Blerick				6 47	Ⓐ 6 54	7 17	⚒ 7 29	7 47	Ⓐ 8 03		
80	**Venlo**	A			Ⓐ 6 50	Ⓐ 6 58	Ⓐ 7 20	⚒ 7 33	7 50	Ⓐ 8 07		
80	**Venlo**		Ⓐ 6 25		⚒ 6 55		7 25		7 55			
85	Tegelen		6 31		7 01		7 31		8 01			
92	Reuver		6 39		7 09		7 39		8 09			
98	Swalmen		6 46		7 16		7 46		8 16			
103	**Roermond**	A	Ⓐ 6 51		⚒ 7 21		7 51		⚒ 8 21			
	Roermond		Ⓐ *6 54*		⚒ *7 24*	⚒ 7 32	⚒ *7 54*	⚒ 8 02	*8 24*	8 32		
	Sittard	A	Ⓐ *7 13*		⚒ *7 43*	7 48	⚒ *8 13*	8 18	*8 43*	8 48		
	Sittard		⚒ *7 21*	1 7 15	*7 51*	7 49	*8 21*	8 19	*8 51*	8 49		
	Heerlen	A	⚒ |	1 7 37	*8 13*	|	|	⚒ 8 36	*9 13*	|		
	Maastricht	A	⚒ *7 41*			⚒ 8 04	*8 41*			9 04		

51 a > vervolg >

km	treinnummer		2027 931	4631 1929	*1827* 6233 *6933*	6132 833	4633	6235 *6835*	4635 1933 935	*1831* 6237 *6937*	6136 837	4637
0	**Arnhem**		Ⓐ 9 17	⚒ 9 21	*9 34*	9 45	9 51	10 06	⚒10 21	*10 34*	10 45	10 51
11	Elst	A	|	|	*9 41*	9 54	|	10 14	|	*10 42*	10 54	|
11	Elst		|	|	*9 41*		|	10 14	|	*10 42*		|
19	**Nijmegen**	A	Ⓐ 9 31	⚒ 9 35	*9 49*		10 05	10 22	⚒10 35	*10 50*		11 05
19	**Nijmegen**				⚒ 9 54			10 24		⚒10 54		
19	Nijmegen Heyendaal				9 57			10 27		10 57		
33	Cuijk				10 06			10 36		11 06		
43	Boxmeer				10 17			10 47		11 17		
50	Vierlingsbeek				10 23			10 53		11 23		
57	Venray				10 32			11 02		11 32		
78	Blerick			10 38	10 47			11 17	11 38	11 47		
80	**Venlo**	A		10 42	10 50			11 20	11 42	11 50		
80	**Venlo**				10 55			11 25		11 55		
85	Tegelen				11 01			11 31		12 01		
92	Reuver				11 09			11 39		12 09		
98	Swalmen				11 16			11 46		12 16		
103	**Roermond**	A			⚒11 21			11 51		⚒12 21		
	Roermond		11 02		*11 24*	11 32		⚒ *11 54*	12 02	*12 24*	12 32	
	Sittard	A	11 18		*11 43*	11 48		⚒ *12 13*	12 18	*12 43*	12 48	
	Sittard		11 19		*11 51*	11 49		*12 21*	12 19	*12 51*	12 49	
	Heerlen	A	11 36		*12 13*	|		|	12 36	*13 13*	|	
	Maastricht	A				12 04		*12 41*			13 04	

1 niet zaterdags, niet op zon- en feestdagen en niet op 11 en 12 feb

Fig. 14.16.

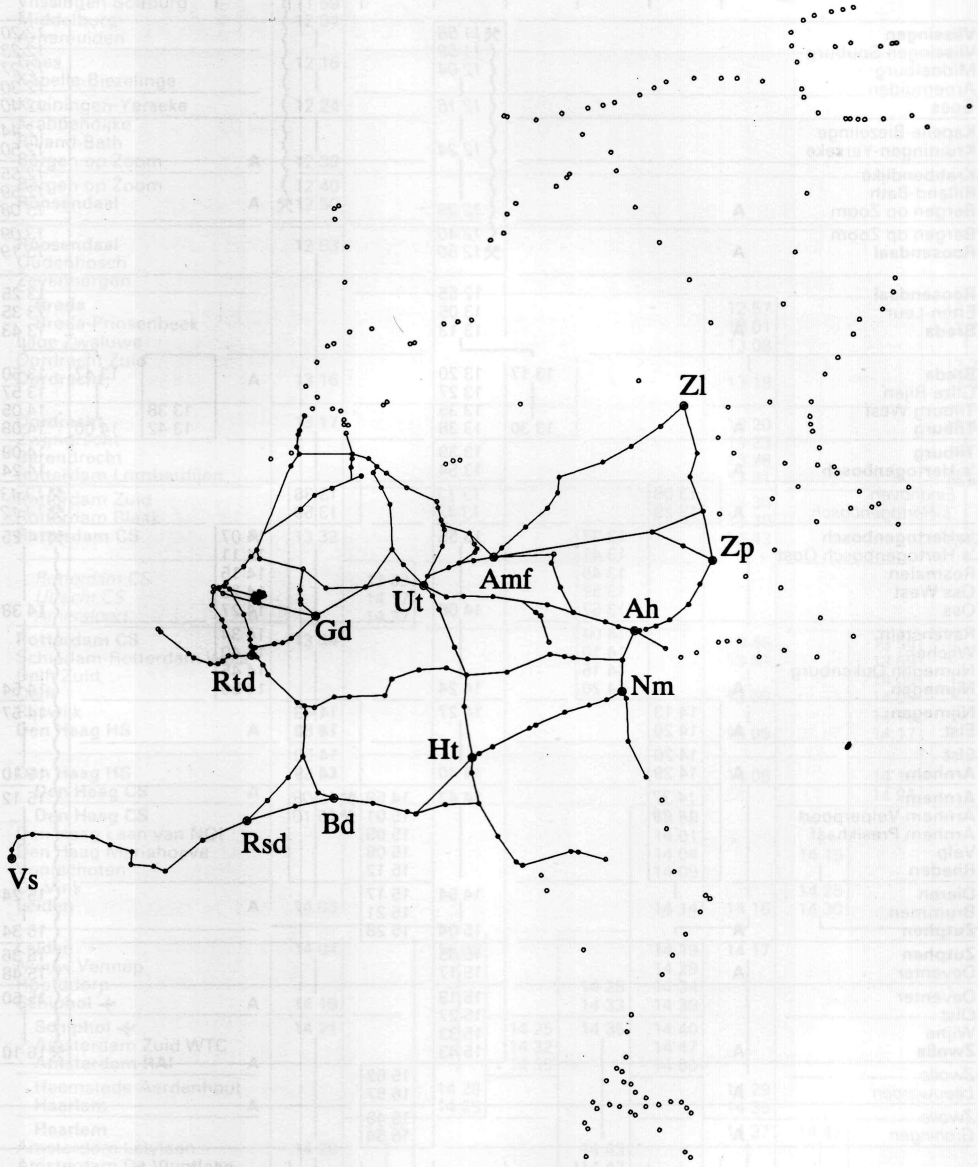
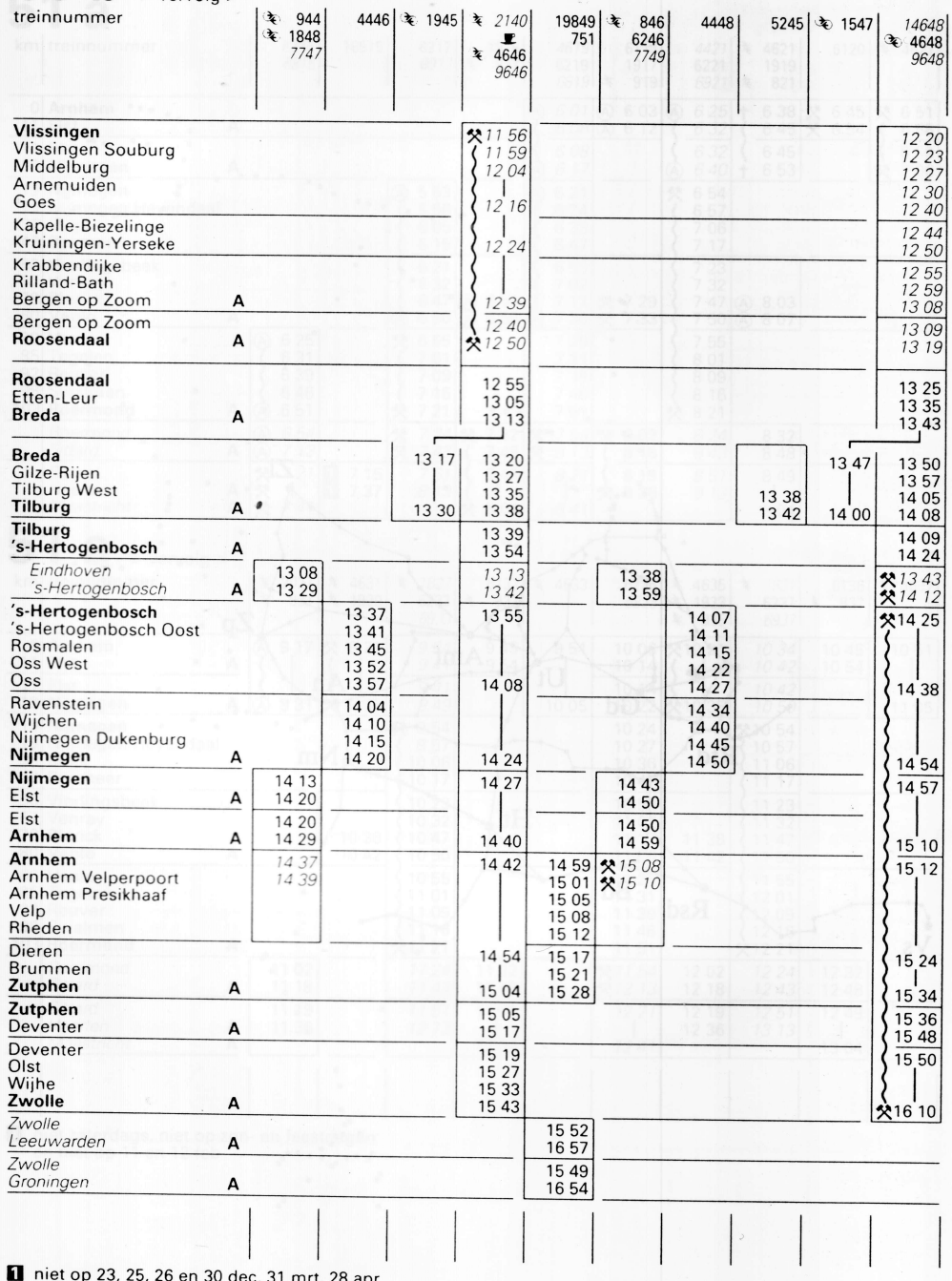


Fig. 14.17. The search space for Vs - Zl

60 b > vervolg >



1 niet op 23, 25, 26 en 30 dec, 31 mrt, 28 apr en 19 mei

Fig. 14.18.

10 b > vervolg >

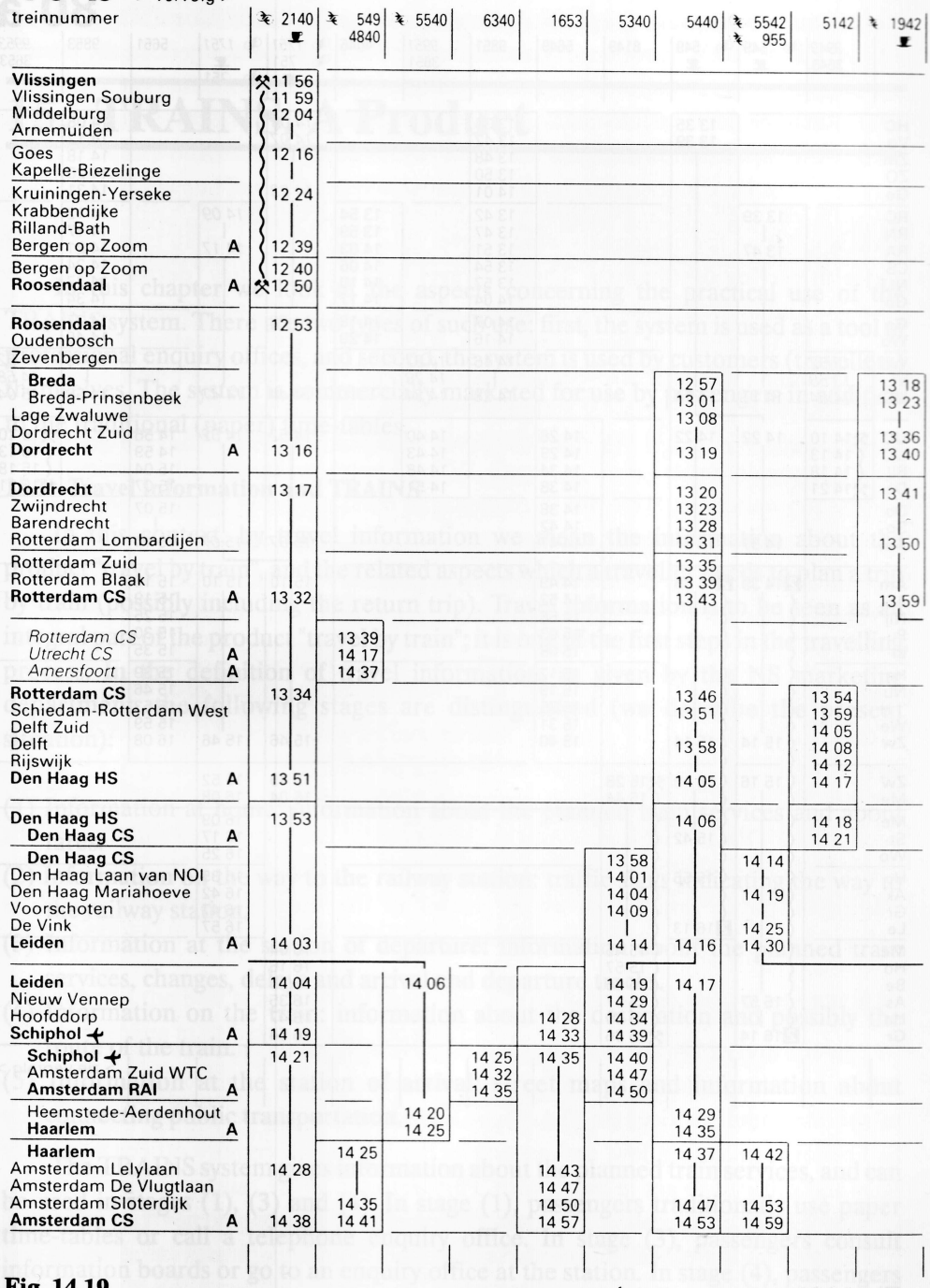


Fig. 14.19.

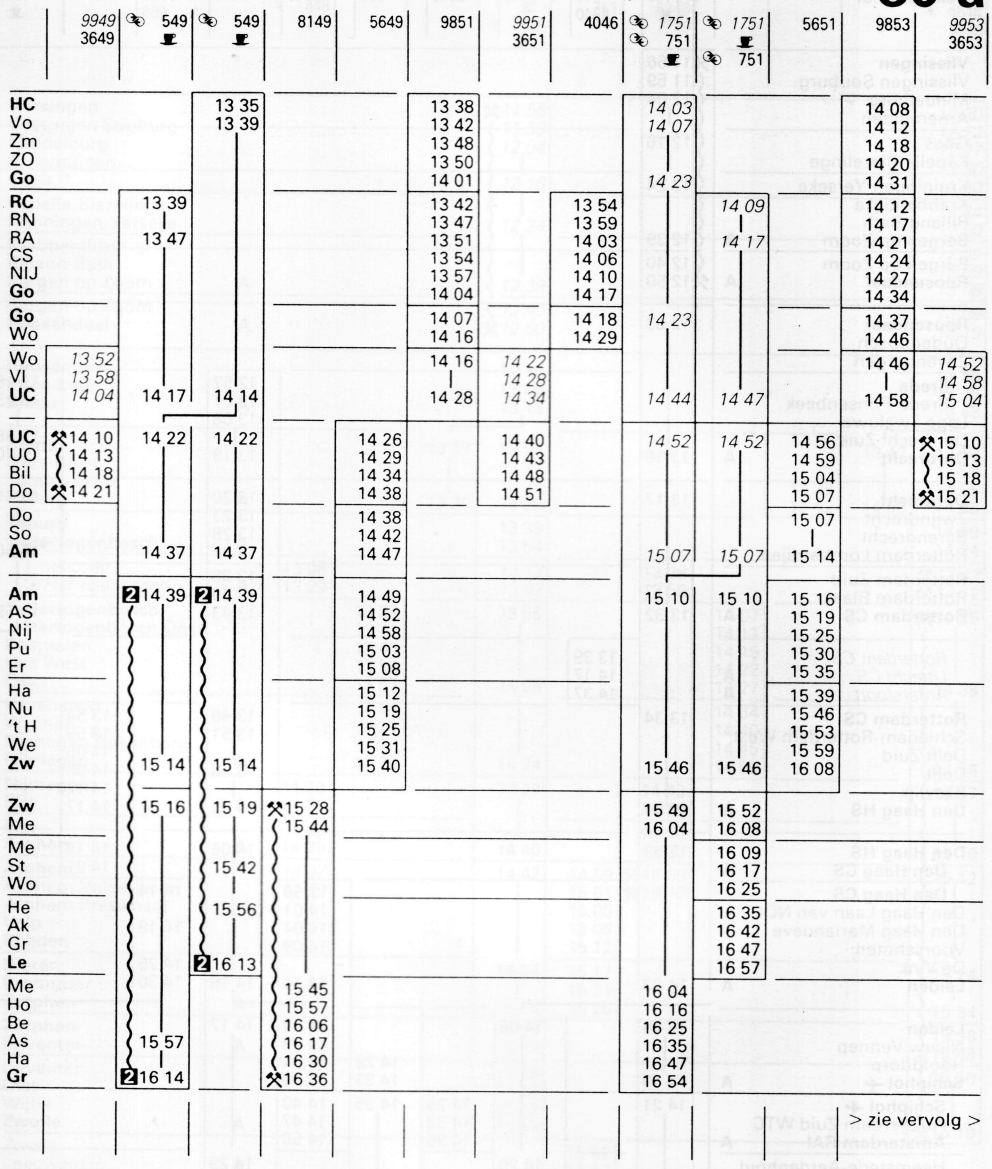

9949 3649
549
549
8149
5649
9851
9951 3651
4046
1751 751
1751 751
5651
9853
9953 3653
HC Vo Zm ZO Go
RC RN RA CS NIJ Go
Go Wo
Wo Vl UC
UC UO Bil Do
Do So Am
Am AS Nij Pu Er
Ha Nu 't H We Zw
Zw Me
Me St Wo
He Ak Gr Le
Me Ho Be As Ha Gr
13 35 13 39
13 39 13 47 14 17
14 14
14 10 14 13 14 18 14 21
14 22 14 37
14 22 14 37
2 14 39 15 14
2 14 39 15 14
15 16 15 57 2 16 14
15 19 15 42 15 56 2 16 13
15 28 15 44 15 45 15 57 16 06 16 17 16 30 16 36
13 38 13 42 13 48 13 50 14 01
13 42 13 47 13 51 13 54 13 57 14 04
14 07 14 16
14 16 14 28
14 26 14 29 14 34 14 38
14 38 14 42 14 47
14 49 14 52 14 58 15 03 15 08
15 12 15 19 15 25 15 31 15 40
13 54 13 59 14 03 14 06 14 10 14 17
14 18 14 29
14 22 14 28 14 34
14 40 14 43 14 48 14 51
14 03 14 07 14 23
14 23 14 44
14 52 15 07
15 10 15 46
15 49 16 04
16 04 16 16 16 25 16 35 16 47 16 54
14 09 14 17
14 23 14 47
14 52 15 07
15 10 15 46
15 52 16 08
16 09 16 17 16 25
16 35 16 42 16 47 16 57
14 56 14 59 15 04 15 07
15 07 15 14
15 16 15 19 15 25 15 30 15 35
15 39 15 46 15 53 15 59 16 08
14 08 14 12 14 18 14 20 14 31
14 12 14 17 14 21 14 24 14 27 14 34
14 37 14 46
14 46 14 58
15 10 15 13 15 18 15 21
14 52 14 58 15 04
> zie vervolg >

Fig. 14.20.

15. TRAINS, A Product

In this chapter we look at the aspects concerning the practical use of the TRAINS system. There are two types of such use: first, the system is used as a tool at professional enquiry offices, and second, the system is used by customers (travellers) themselves. The system is commercially marketed for use by passengers in addition to the traditional (paper) time-tables.

15.1. Travel information and TRAINS

In this context, by travel information we mean the information about the product "travel by train", and the related aspects which a traveller needs to plan a trip by train (possibly including the return trip). Travel information is to be seen as an integral part of the product "travel by train"; it is one of the first steps in the travelling process. In the definition of travel information, as given by the NS marketing department, the following stages are distinguished (we describe the present situation):

(1) Information at home: information about the planned train services and about fares.
(2) Information on the way to the railway station: traffic signs indicating the way to the railway station.
(3) Information at the station of departure: information about the planned train services, changes, delays and arrival and departure tracks.
(4) Information on the train: information about the destination and possibly the route of the train.
(5) Information at the station of arrival: street maps and information about connecting public transportation.

The TRAINS system gives information about the planned train services, and can be used in stages (1), (3) and (4). In stage (1), passengers traditionally use paper time-tables or call a telephone enquiry office. In stage (3), passengers consult information boards or go to an enquiry office at the station. In stage (4), passengers

consult the signs on the train or ask the conductor. The TRAINS system was first used at telephone enquiry offices, later also at enquiry offices at railway stations. Then the system was introduced for home and office use by passengers. In the future, the TRAINS system may also be consulted through Videotex terminals at home, the system may be used in information pillars at stations, and conductors may carry a hand terminal with the system.

The need for travel information is common to all passengers. However, this need is greatest for passengers which travel relatively little (1 to 10 trips per year) and with varying destinations. The NS travel information policy (derived from the strategic and marketing policy, in which travel information plays an important role) is based on the needs of these passengers. If the needs of these passengers are met, then certainly, this will also be the case with the other types of passengers. The TRAINS system is especially suited for passengers who travel to many different destinations. For stage (1) the NS travel information policy has the following objectives:

(1) Travel information should be as much as possible available 24 hours per day.
(2) Travel information should be as much as possible individually oriented.
(3) Travel information should be supplied at a low cost to the customer.

The TRAINS system contributes significantly to the realization of these objectives. When used at an enquiry office or at home, the system will provide answers tailored to the specific wishes of a passenger. And of course, when used at home it is available 24 hours per day. Furthermore the retail price of the TRAINS system (*f* 9,95 , about US $ 5.00) is very low. Because of the added functionality, the ease of use, and because the TRAINS system is used by personnel (at enquiry offices and throughout the NS organization) as well as by the passengers themselves, it has replaced the traditional paper time-tables as the center of the travel information policy, as defined by the NS marketing department.

15.2. TRAINS as a tool at enquiry offices

By the end of 1987, travel information by telephone was provided by 12 enquiry offices at railway stations, by 255 stations without separate enquiry offices, and by 2 telephone enquiry offices (in The Hague and Hengelo). However, the total capacity had become much too low, and especially the telephone enquiry offices were very busy and difficult to get in touch with. Furthermore, at the railway stations without enquiry offices, information was given by unqualified personnel (not trained to give travel information). Due to these facts, the overall quality of travel information was poor and many customers were dissatisfied. In order to improve the quality it was

decided to centralize travel information by telephone into three telephone enquiry offices (in Utrecht, The Hague and Hengelo) which give solely travel information (both planned train services and information on presently running trains). All personnel is trained specifically for giving travel information. In order to improve the quality of information, in the spring of 1988 TRAINS was introduced as a tool at these enquiry offices. The three offices have one single national telephone number (06 - 899 1121) and have 39 telephone lines. In total, 180 people work at the enquiry offices (mostly part-time), currently (September 1990) handling about 2.5 million calls per year. A call is not toll free; the customer pays *f* 0,40 per minute.

15.2.1. Introducing TRAINS at enquiry offices

The users of TRAINS at enquiry offices had no previous computer experience whatsoever. Therefore much effort was put into a careful introduction of the system. A project team was formed for the introduction. Members of this team were the two developers of the system (working for CVI, Centrum Voor Informatieverwerking, a subsidiary company of NS), a team manager from the NS exploitation department responsible for travel information, a representative from the NS exploitation department responsible for the information systems containing the time-tables, a representative from the NS marketing department responsible for travel information, a professional NS instructor, and the heads of the three enquiry offices.

The program was made as much user-friendly as possible. For instance, since most new users could hardly type, a powerful name recognition algorithm was designed to handle misspellings of station names. Since both experienced personnel from previous enquiry offices and new personnel would be working with the system, it had to be made sure that the user interface accommodated both types of users. For instance, a station can be entered by its official abbreviated name, but also by its full name (possibly incomplete or misspelled). The interface was designed in such a way that users can quickly perform the most frequent queries:

(1) What is the best way to travel from one station to another, departing at or after, or arriving at or before some specific time?
(2) What is the appropriate fare?
(3) What are the possibilities for the return trip?
(4) What is an earlier or later train?
(5) What is the earliest or latest possibility?

TRAINS' active component (discussed in chapter 13) makes sure that alternate trip possibilities are found, thereby providing the caller with all relevant information.

Of course, introducing a computer at a site where previously all work was done manually, creates some fear. First, people may fear that their job may become redundant. Second, people may fear that they will not be able to master this new technology and will lose their job. When we started the introduction of TRAINS, we emphasized that the system was a new, easy to use tool to make jobs more pleasant. The users would not have to tediously search the paper time-tables anymore (which were notoriously worn out after a few weeks of service), and have more time to help the client on the telephone, while reducing the time the client would have to wait. We were careful **not** to say that the system would be *better* than they were, just *faster*.

When the introduction was started, the system was not transferred to the enquiry offices. Instead, we left the system in a room at the company headquarters in Utrecht. We asked the heads of the three enquiry offices to come to Utrecht about 4 hours per week to help us develop the system further. With the help of a professional instructor (who was experienced in teaching personnel at enquiry offices and at ticket counters) they learned to operate the system. Although these people would never operate the system in practice, this approach had a number of advantages:

(1) The fact that the heads of the enquiry offices were the first to work with the system (at the company headquarters!) confirmed their status at the enquiry office. The introduction of the system further confirmed their status.
(2) At the company headquarters, away from their personnel, the heads of the enquiry offices were much less inhibited than they would have been at their enquiry offices. They were allowed to make mistakes without becoming embarrassed and without their personnel knowing it. By the time the system was transferred to the enquiry offices, the heads were experienced with the new system, and could reassure and help their personnel themselves. Since the heads of the enquiry offices were experienced with the system before their personnel was, again their status was confirmed instead of endangered by this new technology. They were actively involved in the introduction.
(3) After initial training, the subsequent, additional on the job training of personnel, could be left to the heads of the enquiry offices.
(4) The heads of the enquiry offices were all very experienced and had spent several years giving travel information themselves. Their comments greatly helped to improve the system and especially the user interface (the design and lay-out of the screens, the use of keys, and the refinement of the user model).

During these sessions, there was much direct contact between the developers of the system and the heads of the enquiry offices. Even the smallest comments were discussed with the project team, and the effects would almost always be visible at the

next session. Quickly seeing the effects of their comments, the heads of the enquiry offices gained much confidence in the system and its developers. The system was made *for* them and *with* them. Towards the end of their introduction to TRAINS, they regarded the system as *their* system as well. The disadvantage of this approach was that the user management was only distantly in touch with the development. They did hear about the system, the comments and alterations, but always much later. Still, we feel that if the communication had been through the conventional management channels, too many comments would have been at least much delayed, possibly coloured or even filtered out! We feel that the active involvement of the heads of the enquiry offices and the direct contact between the users at the enquiry offices and the developers of the system have been critical success factors during the introduction.

It must be emphasized that much of the system has been adapted during the introduction and the first year of professional use. Although the underlying (search) algorithms have remained largely the same, practically every other aspect of the system has been revised in a prototyping manner: the users were given a prototype system they could try out, the prototype was adapted to remarks and wishes, then the users tried out the new prototype, and so on. Especially since the users of TRAINS had very little or no experience with information systems, they could not be expected to articulate their requirements completely and clearly. The prototyping method proved to be an excellent way to determine these requirements. For a discussion of the prototyping method see [Je, 1983] and [Da, 1984].

15.2.2. The effects of professional use of TRAINS

After the system was introduced at the enquiry offices and the users there acquired experience with the system, a number of effects became clear. We shall look at a number of them.

15.2.2.1. Speed

The system performs much faster than human beings. Queries are answered in 15 to 30 seconds (when the system runs on Atari ST computers). It is estimated that by using the system, questions about trip possibilities and fares are handled in 50 to 70 percent of the time taken before the introduction of TRAINS. Note that the duration of the complete call is measured. With the TRAINS system, most of the call

consists of dialogue, the silence while the operator is searching has almost completely disappeared. Furthermore, it seems that with the TRAINS system, the operators tend to give more information[1].

15.2.2.2. Quality

In non trivial cases, the system often gives better answers than human beings. The system is unprejudiced and will find the optimal route, no matter how hard it would be to find the route in the paper time-tables, or how unusual the route may seem. We found that people tend to prefer certain routes for specific queries. It turned out that many people working at enquiry offices used 'old' knowledge. The routes they preferred used to be optimal some time ago. However, they did not update their knowledge very well, every time the time-tables were changed. As a result, they would give suboptimal solutions.

15.2.2.3. Consistency

Since the TRAINS system is used throughout the organization and at all enquiry offices, customers will get the same advice everywhere. Previously however, the advice would have depended on the knowledge of the person asked. For instance, the people working at the enquiry office in The Hague have a good knowledge of the possibilities locally, but less so of the possibilities around, say, Hengelo. The people working at the enquiry office in Hengelo would advise to travel from Hengelo to Maastricht via Utrecht (a considerably longer than usual, but equally fast route which is more comfortable, see chapter 13). The people at the enquiry office in The Hague, however, would not have considered that route.

15.2.2.4. Regulations

It became clear that some NS regulations needed to be more precise. For instance, one regulation states that a detour is allowed if it makes a journey quicker or more comfortable. The rule does not give a limit to the detour. Previously, many detours giving better journeys were too hard to find in the time-tables and not many people knew about them (and obviously did not use them). But the TRAINS system will always find a quicker journey, no matter how unusual the route may be or how difficult it may be to find it in the time-tables. It was then decided to let the personnel decide whether the detour was reasonable or not, which introduced a new

1 Mrs. J. Weggemans, head of the enquiry office in Utrecht at the time of the introduction (personal communication).

form of inconsistency. By now, the detours generated by TRAINS are automatically accepted as reasonable. We can say that TRAINS embodies this regulation!

15.2.2.5. Awareness

Apart from the data of the planned train services, the TRAINS system needs detailed information about which possible connections are feasible and which are not. Since the system requires information about **all** possible connections, not just the most important connections (as indicated in the paper time-tables), this placed a higher demand on the information from the time-table planning department. With the TRAINS system they can actually see how their planning works out in practice; TRAINS enables them to tune the time-tables for better connections more easily. In general, the use of the system throughout the NS organization has led to a higher awareness of the quality of the time-tables, which is a major part of the quality of the entire product "travel by train".

15.3. TRAINS as a commercial product

In the summer of 1989, after the TRAINS system had been used successfully at enquiry offices for one year, it was decided that the system could be released to the general public in May 1990.

15.3.1. Releasing TRAINS

Releasing a product like TRAINS does have some risks:

(1) If the system would malfunction or would not satisfy the customers, then that would damage the image of NS and give the organization a bad reputation not easily forgotten.
(2) The system could be misused, for instance by the press to generate negative publicity for NS.

Therefore, in order to try out the reaction from the public and the press, it was decided to have a prerelease during the fall of 1989. Before that, the system was first tested at some external sites, for example at the ANWB (the largest Dutch automobile club) and the Ministry of Transport. When the reactions turned out to be positive, in October 1989, the system (now called "NS Reisplanner"; the NS Travel Planner) was prereleased in limited numbers as a promotional gift of CVI (Centrum Voor Informatieverwerking, the software company of NS), where the system had been developed and which was celebrating its 25th anniversary. In this way, an unsuccessful prerelease would have had only a minor effect on the image of NS. Of

course, it would have been damaging for the image of CVI, but people would have been more forgiving since the system was a promotional gift. Furthermore, the image of a mid-size software company such as CVI is less sensitive than the image of a national railway company such as NS.

Fortunately, the prerelease was successful and due to the limited edition, within a few weeks the system turned into what was called the most frequently copied program of the country. During this period of unofficial use, many people encouraged NS to release TRAINS as an official product and useful suggestions were made on how the system could be improved.

We shall now look at the aspects concerning the release of the TRAINS system as an official NS product.

15.3.1.1. The product

The primary objective of the product is to distribute the planned NS train services. A secondary objective has been to enhance the NS image as a modern company. The product is aimed at the following categories of users:

(1) Individual travellers.
(2) Companies, business travellers.
(3) Public information suppliers such as libraries and tourist information offices.

Since the same system would be used by customers as well as professional users (NS employees), it had to be easy to use for both types of users. In case of a compromise, it was decided that the emphasis should always be on the customer. Therefore, for instance no jargon was allowed in the program. Since the user interface was designed already to accommodate unexperienced NS users, only minor changes were necessary. The use of (the official) abbreviated station names was made optional (by default not available). Since the train numbers are only useful for (NS) internal reference, they too were made optional (by default not shown). Furthermore, additional information about the tariff distances of the routes the system finds (and which are used to calculate fares) were made optional (not shown by default). By making optional these features, which are primarily aimed at professional use, and by supplying a user profile facility, both types of users are accommodated. The functionality of the system was not changed; all information available to professional users are available to the customers. The system was made available for MS DOS and Atari ST computers.

15.3.1.2. Price

It was decided to give the system a retail price which was as low as possible (without making a loss) for two reasons. First, the NS travel information policy is to supply information at a low cost to the customer. Second, selling information is not viewed as an NS activity. Giving travel information is seen as a service aspect and as an integral part of the NS product "travel by train". Because the system offers more service than the conventional time-tables (retail price *f* 6,50), it should be priced accordingly. Since the system was originally written for professional use, the development costs were not taken into account, and only the costs of production, distribution and promotion (the "out of pocket" costs) needed to be covered. This way, a retail price of *f* 9,95 was calculated.

15.3.1.3. Packaging

An attractive cassette was designed to contain a floppy disk (both the 3.5 and 5.25 inch formats can fit in the cassette), a manual and a railway map. The lay-out and colours (of the cassette, the manual and even the floppy disks themselves) were chosen to fit the NS corporate style. The objective was to make the package immediately recognizable as an NS product. A link to the paper time-tables was made by using the cover art of the (paper) time-tables for the cassette of the floppy disk.

15.3.1.4. Promotion and distribution

The distribution of the product was kept limited for three reasons:

(1) NS had no previous experience with selling software.
(2) The potential of the product was unclear since commercial software of such a low cost was unprecedented.
(3) Since three versions of the system were available (on 3.5 inch and 5.25 inch floppy disks for MS DOS computers, and on 3.5 inch floppy disks for Atari ST computers) an extensive distribution would be complicated and costly.

It was decided to mainly distribute the system by mail order and to sell it in special NS shops ("Spoorwinkel") at the railway stations of Amsterdam CS, Utrecht CS and The Hague CS, and at a mobile promotion stand. Distribution by a third party was not chosen since that would have given too little control over the distribution and would have added too much to the price.

Due to a limited budget the promotion was kept modest. It was made part of the "monthly marketing theme". The marketing theme of May 1990 was the new train services and the release of the system was part of that. In a brochure about the new train services, the system as well as the new time-tables were announced and an order form for the system was enclosed. These brochures were available at all ticket counters. Furthermore, the system was mentioned in the new time-tables and in a brochure for regular customers (again including an order form). There was no separate promotion campaign. Review copies of the system were sent to computer magazines. In November 1990, the TRAINS system was featured in an NS "corporate brochure", which described 8 important NS projects for the nineties, and which was aimed at enhancing the NS image as a modern company. This brochure was backed by a poster campaign at railway stations.

15.3.1.5. Personnel

The general NS personnel was made aware of the new product by a small announcement in an article about the new train services in the newsletter sent to all NS personnel. An extra announcement was made in the special news bulletin for retail personnel, explaining the distribution matters and the specific (customer) brochure with order form.

15.3.2. Sales and effects

The system went on sale as an official NS product on May 5, 1990. Although the system seems to be copied extensively (again!), the sales have been higher than expected: 35 000 copies over the first 6 months! The main reasons are probably the low price and the attractive packaging. At the end of 1990, we estimate that there are about 100 000 copies of the system in use. Apparently the system performs very satisfactorily. We have received many compliments and only very few complaints were made, and almost all of them about faulty disks. The only criticism has been about the limited distribution. The most frequent suggestions have been about adding platform information and a digital map.

The first effect of the sale of the system that could be seen, has been a decrease in calls to telephone enquiry offices. In an early internal NS study (August 1990) it has been calculated that statistically, each floppy disk sold gives a mean decrease of 6 telephone calls per year. There has been no clear effect on the sale of paper time-tables. Many customers bought both the program and the paper time-tables.

An important result of the high sales figures is that unscheduled updates of the system, to deal with changes in the time-tables in the course of the year, would be

costly and complicated. Normally, the system is valid one year, and each year a new release with the new time-tables is sold again. An update for systems sold the previous year is not offered. Due to the low cost of the software, this is considered acceptable by the customers. Multiple releases per year would probably not be considered acceptable, however. If the time-tables would be changed considerably in the meantime, an update of all systems sold would be costly. If an unscheduled important change of time-tables is considered, this will have to be taken into account.

15.3.3. Alternate use

Apart from being used to plan a trip by train, TRAINS is also used for other purposes. In Amsterdam, the system is being used in a labour council project to introduce long term unemployed people to computers and to take away fear of computers (apparently, the user interface design meets the objectives that were set). At the university of Leiden, the system is used in a first session of an introductory course "Law and Informatics" for law students. In 1991, the system will also be used in information science lessons at secondary schools throughout the country. Many companies use the fares information of the system to check trip expense accounts. Personnel departments use the system to advise employees and applicants on how to travel to the company site.

15.4. The future

In the future, the system will remain in use at enquiry offices and each year a new version with the new train services will be released to the public. Over the next years, new features will be added to increase the service and to keep the interest of the public (for instance digital maps and platform information). Apart from versions giving information about NS (national) train services, there may also be versions giving information about (some) international train services.

NS is currently developing new retail systems which will be used at ticket counters (making the different types of tickets and handling the cash register functions). The TRAINS system will also be included in these new systems in order to provide customer information. The system may also be used in information pillars and ticket machines. The system is currently used in an experimental Videotex application. We will be investigating the possible use of TRAINS in the planning process of new train services.

Transport by train, however, is only part of the total public transportation system and it has become clear that a system giving information about all forms of public

transportation is a necessity. Such a system is now under development (see chapter 16).

Finally, we are currently also developing versions of TRAINS for foreign railway companies and versions including airline information. A prototype version of TRAINS including both railway and airline services was recently shown to airline companies.

16. Further Developments

In this chapter we shall first look at some recent further developments of the TRAINS system, and then discuss some future work. This chapter is largely similar to [Tu, 1990].

We recently adapted the TRAINS system to include not only train services, but also other forms of public transportation. Our first step was to introduce regional buses. The next step was to include intra-city transportation, such as buses, trams and subways. Currently, we are building a system which will include the entire Dutch public transportation system. This information system will be fully operational in 1992.

16.1. Representing other forms of public transportation

In order to ensure high quality information, we had to extend our representation of transportation services. The concept of time-tables is common to all forms of public transportation. The frequency and the punctuality differ, however. The frequency does not affect the quality of time-table information, but the punctuality does. If the punctuality is low, time-table information is not of much use. Trains, subways and regional buses are both fairly punctual. Intra-city transportation like tramways and buses, has a relatively low punctuality because one heavily used infrastructure is shared with other city traffic. In practice, the time-tables are used as a reference only. In order to give useful information, we dropped the discrete departure times and travel times for these types of transportation, and replaced them by an estimated waiting time (at the stop) and an estimated travel time. These times are chosen conservatively and may depend on the time of day (the delays may be greater during rush hours). In this way, the system will give a realistic advice.

16.2. Discontinuities in public transportation services

As described in chapter 13, the active component of the TRAINS system searches for discontinuities (resulting in alternate journeys) by trying different times of departure and arrival. However, not only changes in the desired time of departure

or arrival may cause discontinuities: the choice of station may also have effects. For instance, at the (railway) station Amsterdam Lelylaan all (fast) intercity train services stop, while at the station Amsterdam De Vlugtlaan (about 2 km from Lelylaan) only the (slow) stopping trains stop. It may be more efficient to take more time to go to the Lelylaan station instead of the station De Vlugtlaan, and then take advantage of the intercity services.

Usually, situations in which the choice of station is relevant are rare in a national railway network. But when we consider intra-city transportation services, these situations are very common. Many bus and tram stops are within walking distance from each other (providing connections), or multiple stops are at a more or less equal distance from a destination. Favouring a certain stop over another one may have different reasons:

(1) A stop may be serviced by faster links.
(2) A stop may be more frequently serviced.
(3) A stop may be serviced by a link which provides quicker connections.
(4) A stop may be serviced by a link providing a more direct connection.

In the first three cases the travel time of a complete journey (which may consist of multiple stages and different modes of transport) will decrease. The last case may provide a more convenient journey.

16.2.1. The human solution: maps

Humans deal with stops which are near to each other by using maps. They look for stops in the neighbourhood of a certain stop and assess the advantages of the different possibilities. If the destination of a trip does not have a stop in the immediate proximity, then this situation is handled similarly: on the map the different stops in the neighbourhood are assessed. Humans, however, seldomly have complete and unbiased knowledge of the different possibilities and therefore the solution which is found is often suboptimal. This is particularly the case if local knowledge is lacking or limited. For instance, for a trip to Amsterdam, many people choose Amsterdam Central Station because they think that the biggest station will provide the best connections. However, Amsterdam has 7 more stations, some of which might be much better suited considering the time-tables or the geographical location of the exact goal of the trip (within Amsterdam).

16.2.2. The computer solution: digital maps

In order to extend the active behaviour of our system we use the same approach as the human solution in combination with complete knowledge. We use digitized

map information. Of each stop in the network the exact geographical position is known.

16.2.2.1. Estimating walking distances

By using the coordinates of the stops and simple two-dimensional geometry, the distance between two stops can be estimated. For the computation of this estimate we use the *Manhattan* distance: all stops are assumed to lie on a grid of straight roads with 90° angled corners, much like a street map of Manhattan Island (New York City). By using an estimate of the walking speed (in km/h), we can estimate the time to walk the distance.

16.2.2.2. Choosing the stops

When the user of the system enters the name of a stop (either the place of departure or arrival), the system will search in the neighbourhood of this stop for possible alternate stops. All stops within a walking distance of 10 minutes are considered. This selection is suggested to the user who may remove stops from this selection or add extra stops. Then *all* stops in the selection are considered in the search for possible journeys. Each stop is given a penalty equal to the walking distance. If there are 6 possible departure stops and 5 possible arrival stops, then all 30 possible journeys are evaluated. The best solutions in terms of travel time and number of changes (train changes, bus changes etc), taking into account the walking distances, are selected and suggested to the user.

16.2.2.3. Special objects

Sometimes, a user has very limited local knowledge. She only knows the address of where she would like to go, or not even that: she would like to go to a specific museum of which she does not know the exact address. For these cases we have introduced *special objects*. A special object can be a district name, a street name, a postal code, the name of a shopping center, a museum, a hospital, etc. The exact geographical position is known for each special object. When a special object, such as a postal code, does not have a single position, then the geographical position of its center is used. Long streets may be subdivided into different parts (which are distinguished by house numbers). By using the same approach as for determining alternate stops, stops in the neighbourhood of a special object are determined.

16.2.2.4. Obstacles

Although two stops may be very close geographically, it may not be possible to walk from one stop to the other. For example, they may be separated by a canal or a river. For these situation we have introduced *obstacles*. The position and the length are known for each obstacle. By using two-dimensional geometry it can be computed how much time it would take to walk past the obstacle.

16.2.2.5. The advantage of using geographical information

The paramount advantage of using the coordinates of the geographical location of a stop or an object, is that this approach combines a high degree of flexibility and maintainability. Stops or objects can be added or removed very easily, no extra information about other stops or special objects nearby is needed. For input, no relation between stops and objects near the new stop need to be given. The system will objectively decide which stops are near, without bias and with complete information.

16.3. Combining time-tables and geographical information

By using complete information of both time-tables and geographical locations of stops, the system is capable of giving high quality information: the best possibilities considering time and space are found. Alternatives are not only found by trying different times, but also by trying different stops which are near the origin or destination. With complete knowledge and without bias the best suited journey is found.

16.4. Using TRAINS in the time-table planning process

In the future we shall be investigating the possible use of the TRAINS system in the time-table planning process. An important step in the time-table planning process is the evaluation of a proposed time-table. By supplying a test set of trips (origin and destination pairs), the TRAINS system can be used on the new time-table to compute the resulting travel times. By weighing the different origin and destination pairs, and by comparing the resulting travel times to the optimal travel times (which are direct connections without stops), an indication of the quality of the time-table can be obtained. Furthermore, by using information about the numbers of travellers, an indication of the load of the different lines can be obtained.

References

[Be, 1958] Bellman, R.E., "On a routing problem", *Quarterly of Applied Mathematics*, **16**, 1958, pp. 87 - 90.

[Bu, 1968] Butas, L., "A directionally oriented shortest path algorithm", *Transportation Research*, **2**, 1968, pp. 253 - 268.

[Cl, 1972] Clercq, F. Le, "A public transportation assignment method", *Traffic Engineering and Control*, June 1972.

[Co, 1966] Cooke, K.L. and Halsey E., "The Shortest Route Through a Network with Time-Dependent Internodal Transit Times", *Journal of Mathematical Analysis and Applications*, **14**, 1966, pp. 493 - 498.

[Da, 1960] Dantzig, G.B., "On the Shortest Route through a Network", *Management Science*, **6**, 1960, pp. 187 - 190.

[Da, 1966] Dantzig, G.B., "All Shortest Routes in a Graph", *Technical Report 66-3*, Operations Research House, Stanford University.

[Da, 1977] Dawson, C., Siklóssy, L., "The role of preprocessing in problem solving systems", *Proceedings of the fifth International Joint Conference on Artificial Intelligence*, Cambridge, Ma, 1977, pp. 465 - 471.

[Da, 1984] Davis, G.B., and Olson, M.H., "Management Information Systems", Second edition, Singapore: McGraw-Hill, 1984, pp. 567 - 670.

[De, 1979] Denardo, E.V. and Fox, B.L., "Shortest-route Methods: 1. Reaching, Pruning, and Buckets", *Operations Research*, **27**, 1979, pp. 161 - 186.

[De, 1984], Deo N. and Pang C., "Shortest-path algorithms: taxonomy and annotation", *Networks*, **14**, 1984, pp. 297 - 323.

[Di, 1969] Dial, R.B., "Algorithm 360: Shortest Path Forest with Topological Ordering", *Communications of the ACM*, **12**, 1969, pp. 632 - 633.

[Di, 1979] Dial R.B. et al, "A computational analysis of alternative algorithms and labelling techniques for finding shortest path trees", *Networks*, **9**, 1979, pp. 215 - 248.

[Di, 1959] Dijkstra, E.W., "A Note on Two Problems in Connection with Graphs", *Numerische Math.*, **1**, 1959, pp. 269 - 271.

[Dr, 1969] Dreyfus, S.E., "An Appraisal of Some Shortest-Path Algorithms", *Operations Research*, **17**, 1969, pp. 395 - 412.

[Ev, 1979] Even, S., "Graph Algorithms", Potomac, Md: Computer Science Press, 1979.

[Fa, 1967] Farbey, B., Land A. and Murchland J., "The cascade algortihm for finding all shortest distances in a directed graph", *Management Science*, **14**, 1967, pp. 19 - 28.

[Fl, 1962] Floyd, R.W., "Algorithm 97, shortest path", *Communications of the Association of Computing Machinery*, **5**, 1962, pp. 345.

[Fo, 1956] Ford, L.R., "Network Flow Theory", The Rand Corporation, 1956.

[Ga, 1984] Gallo, G. and Pallottino S., "Shortest path methods in transport networks", in *Transportation Planning Models*, Amsterdam, Elsevier Science Publishers, 1984, pp. 227 - 256.

[Gi, 1973] Gilsinn J., and Witzgall C., "A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees", National Bureau of Standards Technical Note 777, 1973.

[Gl, 1984] Glover, F. et al., "New polynomially bounded shortest path algorithms and their computational attributes", University of Texas, Austin, Center for Business Decision Analysis, 1984.

[Go, 1976] Golden, B., "Shortest-Path Algorithms: A Comparison", *Operations Research*, **24**, 1976, pp. 1164 - 1168.

[Ha, 1968] Hart, P., Nilsson, N.J. and Raphael, B., "A formal basis for the heuristic determination of minimum cost paths", *IEEE Trans. Systems Man Cybernet*, **4**, 1968, pp. 100 - 107.

[Ha, 1974] Harris, L.R., "The heuristic search under conditions of error", *Artificial Intelligence*, **5**, 1974, pp. 217 - 234.

[Hi, 1986] Hillier, F.S., and Lieberman, G.J., "Introduction to Operations Research", Fourth edition, Oakland, Ca: Holden-Day Inc, 1986, pp. 337 and 501 - 502.

[Hu, 1967] Hu, T.C., "Revised Matrix Algorithms for Shortest Paths", *SIAM Journal on Applied Mathematics*, **15**, 1967, pp. 207 - 218.

[Hu, 1968] Hu, T.C., "A Decomposition Algorithm for Shortest Paths in a Network", *Operations Research*, **16**, 1968, pp. 91 - 102.

[Je, 1983] Jenkins, A. Milton, "Prototyping: A Methodology for the Design and Development of Application Systems", Working Paper, School of Business, Indiana University, Bloomington, 1983.

[Ka, 1982] Katoh, N., Ibaraki T. and Mine H., "An efficient algorithm for *k* shortest simple paths", *Networks*, **12**, 1982, pp. 411 - 427.

[Ki, 1969] Kirby R.F. and Potts, R.B., "The minimum route problem for networks with turn penalties and prohibitions", *Transportation Research*, **3**, 1969, pp. 397 - 408.

[Kl, 1964] Klee, V., "A string algorithm for shortest paths in a directed network", *Operations Research*, **12**, 1964, pp. 428.

[Lu, 1989], Luby, M. and Ragde, P., "A Bidirectional Shortest-Path Algorithm with Good Average-Case Behaviour", *Algorithmica*, **4**, 1989, pp. 551 - 567.

[Me, 1984], Mérõ, L., "A heuristic search algorithm with modifiable estimate", *Artificial Intelligence*, **23**, 1984, pp. 13 - 27.

[Mi, 1957], Minty, G., "A comment on the shortest route problem", *Operations Research*, **5**, 1957, pp. 724.

[Mo, 1957] Moore, E., "The Shortest Path through a Maze", *Proceedings of the International Symposium on the Theory of Switching*, Cambridge, Ma, 1957.

[Mu, 1967] Murchland, J.D., "The 'once through' method of finding all shortest distances in a graph from a single origin", *Transport Network Theory Unit Report LBS-TNT 56*, London Graduate School of Business Studies, London, 1967.

[Ni, 1969] Nicholson, T.A.J., "Finding the shortest route beteen two points in a network", *Computer Journal*, **9**, 1969, pp. 275.

[Ni, 1971] Nilsson, N.J., "Problem-Solving Methods in Artificial Intelligence", New York: McGraw-Hill, 1971.

[Ni, 1980] Nilsson, N.J., "Principles of Artificial Intelligence", Palo Alto, Ca: Tioga, 1980.

[Pa, 1974] Pape, U., "Implementation and Efficiency of Moore Algorithms for the Shortest Route Problem", *Mathematical Programming*, **7**, 1974, pp. 212 - 222.

[Pa, 1984] Pallottino, S., "Shortest path methods: complexity, interrelations and new propositions", *Networks*, **14**, 1984, pp. 257 - 268.

[Pe, 1979] Pearl, J., "Studies in heuristics. Part 1: Three variations on a theme in A*", *UCLA-ENG-CSL-7934*, University of California, Los Angeles, Ca, 1979.

[Pe, 1986] Perko, A., "Implementation of algorithms for k shortest loopless paths", *Networks*, **16**, 1986, pp. 149 - 160.

[Po, 1970] Pohl, I., "Heuristic search viewed as path finding in graphs", *Artificial Intelligence*, **1**, 1970, pp. 193 - 204.

[Po, 1971] Pohl, I., "Bidirectional Search", *Machine Intelligence*, **6**, 1971, pp. 127 - 140.

[Po, 1960] Pollack, M. and Wiebenson, W., "Solution of the shortest route problem: a review", *Operations Research*, **8**, 1960, pp. 224 - 230.

[Sa, 1974] Sacerdoti, E.D., "Planning in a hierarchy of abstraction spaces", *Artificial Intelligence*, **4**, pp. 145 - 180.

[Sh, 1976] Shier, D.R., "Iterative methods for determining the *k* shortest paths in a network", *Networks*, **6**, 1976, pp. 205 - 229.

[Sh, 1979] Shier, D.R., "On algorithms finding the *k* shortest paths in a network", *Networks*, **9**, 1979, pp. 195 - 214.

[Si, 1978] Siklóssy, L., "Impertinent Question-Answering Systems: Justification and Theory", *Proceedings of the ACM Annual Conference*, 1978, pp. 39 - 44.

[Si, 1991] Siklóssy, L., and Tulp, E., "The Space Reduction Method", *Information Processing Letters*, **38**, 1991, pp. 187 - 192.

[St, 1974] Steenbrink, P.A., "Optimization of transport networks", New York: Wiley, 1974, pp 150 - 171.

[Tu, 1988] Tulp, E. and Siklóssy, L., "TRAINS, An Active Time-table Searcher", *Proceedings of the 8th European Conference on Artificial Intelligence*, Munich 1988, Pitman Publishing, London, pp. 170 - 175.

[Tu, 1989] Tulp, E. and Siklóssy, L., "TRAINS, A Case Study of Active Behaviour", *Proceedings of the International Workshop on Industrial Applications of Machine Intelligence and Vision*, Tokyo 1989, pp. 259 - 263.

[Tu, 1990] Tulp, E., Verhoef M.L. and Tulp W.L., "TRAINS, Further Implementation of Active Behaviour", *Proceedings of COGNITIVA 90*, Madrid 1990, pp. 711 - 714.

[Vu, 1988] Vuren, T. van and Jansen, G.R.M., "Recent developments in path finding algorithms: a review", *Transportation Planning and Technology*, **12**, 1988, pp. 57 - 71.

[Vl, 1978], Vliet, D. van, "Improved Shortest Path Algorithms for Transport Networks", *Transportation Research*, **12**, 1978, pp. 7 - 20.

[Wa, 1985], Wahlster, W.,, "Cooperative Access Systems", *Future Generation Computer Systems*, **1**, 2, pp. 103 - 111.

[Wa, 1987], Warburton, A., "Approximation of pareto optima in multiple-objective shortest path problems", *Operations Research*, **35**, 1987, pp. 70 - 79.

[Wh, 1960] Whiting, P.D. and Hillier, J.A., "A method for finding the shortest route through a road network", *Operational Research Quarterly*, **11**, 1960, pp. 37 - 40.

[Wh, 1982] White, D.J., "The set of efficient solutions for multiple objective shortest path problems", *Computer Operations Research*, **9**, 1982, pp. 101 - 107.

[Wi, 1984] Winston, P. H., "Artificial Intelligence", Reading, Ma: Addison Wesley, 1984.

[Ye, 1968] Yen, J.Y., "Matrix Algorithm for Solving All Shortest Routes from a Fixed Origin in the General Networks", *Proceedings of the Second International Conference on Computing Methods in Optimization Problems*, San Remo 1968.

[Ye, 1970] Yen, J.Y., "An algorithm for finding shortest routes from all source nodes to a given destination in general networks", *Quarterly of Applied Mathematics*, **27**, 1970, pp. 526 - 530.

[Ye, 1971] Yen, J.Y., "Finding the *k*-shortest, loopless paths in a network", *Management Science*, **17**, 1971, pp. 712 - 716.

Summary

Searching time-table networks

In this thesis we describe an application of AI search techniques to an important class of problems that arise in transportation system analysis. Specifically, this thesis deals with path search problems in space-time networks, a problem commonly arising in connection with scheduled service modes. An important example of a scheduled service mode is a railway transportation service. Apart from a search procedure, we also present a novel representation of the problem domain, and a practical application of the techniques described.

Rather than to adapt a conventional graph representation in order to represent a time-table network, we introduce discrete and discrete dynamic networks for this purpose. In a *discrete network* there are only finite, discrete, predetermined possibilities for moving from one vertex to another. If we consider a railway service network as an example of a time-table network, in a discrete network the stations are represented by vertices, and each train is represented by one connection between vertices. Instead of representing the discrete nature of the scheduled connections (the departure times of the trains) by a function giving the (varying) travel time and wait time of a connection, the connections themselves are made discrete. Each connection representing a train has a discrete start value and end value, representing the time of departure and time of arrival respectively. In a *discrete dynamic network*, in addition, visiting a vertex has a cost (possibly zero), which may depend on both the past and the future route of the path through the vertex. The visiting cost represents the required connectional margin which depends on both the arriving train and the departing train. Furthermore we introduce *dynamic networks*, which lack the discreteness of connections, but in which visiting a vertex has a cost.

In discrete and discrete dynamic networks, due to the discrete nature of the connections, the definition of an optimal path must be adapted: not only has the optimal path the smallest end value (earliest possible arrival time), but also the greatest possible start value, given this end value (the latest possible departure,

given the time of arrival). In order to find such an optimal path, with Dijkstra's algorithm in mind, we have developed a two-pass algorithm for searching a discrete network. Due to the varying visiting costs in a discrete dynamic network and a dynamic network, the Markov independence of optimal solutions is no longer true. This means that an optimal solution for the total problem cannot be constructed by combining optimal solutions of the subproblems. "Divide and Conquer" fails. In a railway service network, the optimal route from A to C via B may not be a combination of the optimal route from A to B and the optimal route from B to C, since there may be no connection. Therefore none of the traditional shortest path algorithms could be used. We have adapted the two-pass algorithm for searching a discrete network to handle discrete dynamic and dynamic networks: we define which solutions to the subproblems are required to be able to construct an optimal solution for the total problem.

In order to increase search efficiency we have developed the Space Reduction Method. In SRM first solutions in a simpler search space, called the abstraction space, are considered in order to cut parts of the entire search space. In a railway service network, the abstraction space consists of a network in which all trains between two stations are replaced by one connection with an estimated travel time. By searching this abstract network it is determined which part of the time-table network is likely to contain the optimal solution. SRM reduces the search space without losing optimal solutions. Once a solution has been found SRM checks whether a better solution might exist outside the reduced search space. We show how SRM can be applied to searching a discrete dynamic network.

Heuristics can be used to further improve the efficiency of search algorithms. We describe how the results from SRM can be used in an A* type of extension to the algorithm for searching discrete dynamic networks, by preferring vertices which are estimated to be closest to the goal during search. SRM gives for every vertex (station) a consistent underestimate of the distance (travel time) remaining to the goal vertex (station of arrival). In the search process, this estimate is used to select the vertices which are estimated to be closest to the goal.

An excellent way to decrease the amount of search necessary to find a solution, is to make sure that the network that is being searched is as small as possible. Some vertices can be removed from the network when they are neither the source, nor the goal vertex. In a railway service network, if it is useless to change trains at a station and if this station is neither the station of departure nor the station of arrival, then we do not need to consider this station when searching. We show how the algorithm for searching discrete dynamic networks can be adapted to deal with these 'hidden' vertices.

When a system is being used to advise travellers about their trip by train, giving only the quickest route is not sufficient. In any practical application travellers also want to know about routes with as few train changes as possible. We describe how the quickest route can be optimized for train changes, and how some longer (suboptimal) routes with fewer train changes can be found by using a time penalty for train changes. By penalizing routes with train changes during search, routes without, or with fewer train changes can be found.

Although the optimal or quickest solution is thoroughly defined, it is far less clear what is the *best answer* to a user's question. In practice, it turns out that users usually overspecify their question and that this question is seldomly definite. There are many factors which determine the 'best' answer, and most users cannot even make all of these factors explicit. Of a trip by train it is known that the number of train changes is important, but there may be additional factors contributing to the best answer. Furthermore, these factors may differ from case to case. Therefore, it is not possible to define the best answer in terms of goals and constraints. For example, in order to find the best answer we cannot just have chosen to use such techniques as multiple-objective shortest path techniques, or techniques to find suboptimal paths for each objective. Instead, we search for a number of optimal solutions, and suboptimal solutions with fewer train changes, and use a general "common sense" user model to select all relevant solutions for a user. The user decides which solution is best for her.

The algorithm for searching discrete dynamic networks and the techniques described previously have been implemented in a working system (TRAINS) which searches the entire Dutch railway service network. We describe how TRAINS was introduced as a tool at information centers of the Dutch railway company NS (Nederlandse Spoorwegen). Subsequently, TRAINS was adapted for public use and released, first as a promotional gift, then as an official NS product (NS Reisplanner).

Samenvatting

Het doorzoeken van dienstregelingsnetwerken

In dit proefschrift behandelen we een toepassing van AI-zoektechnieken op een belangrijke klasse van problemen bij de analyse van transportsystemen. In het bijzonder behandelt dit proefschrift kortste-pad problemen in tijd-ruimte netwerken. Dit soort problemen doet zich in het algemeen voor bij transport op basis van een dienstregeling. Een belangrijk voorbeeld daarvan is een treindienstregeling. Behalve een zoekprocedure, introduceren we ook een nieuwe representatie van het probleemgebied, en een praktische toepassing van de beschreven technieken.

In plaats van een conventionele graafrepresentatie aan te passen om een dienstregelingsnetwerk te representeren, introduceren we discrete en discrete dynamische netwerken. In een *discreet netwerk* is er een eindig aantal discrete, vastgestelde mogelijkheden om van één punt naar een ander te gaan. In plaats van de discrete aard van de dienstregelingsverbindingen te representeren door een functie die de (variërende) reistijd en wachttijd van de verbinding geeft, maken we de verbindingen zèlf discreet. Elke verbinding heeft een discrete begin- en eindwaarde, die respectievelijk de vertrek- en aankomsttijd representeren. In een *discreet dynamisch netwerk* zijn er bovendien kosten verbonden aan het bezoeken van een punt (mogelijk 0 kosten), die kunnen afhangen van zowel de reeds gevolgde als de toekomstige route van het pad via dit punt. Deze bezoekkosten representeren de benodigde overstaptijd. Verder introduceren we *dynamische netwerken*, waarin de verbindingen niet discreet zijn, maar waarin het bezoeken van een punt kosten geeft.

In discrete en discrete dynamische netwerken moeten we door de discrete aard van de verbindingen de definitie van een optimaal pad aanpassen: een optimaal pad heeft niet alleen een zo laag mogelijke eindwaarde (lees: een zo vroeg mogelijke aankomst), maar ook een zo groot mogelijke beginwaarde, gegeven deze eindwaarde (lees: een zo laat mogelijk vertrek, gegeven de aankomsttijd). Om een dergelijk optimaal pad te vinden hebben we, met Dijkstra's algorithme in gedachten,

een uit twee slagen bestaand algorithme ontwikkeld om discrete netwerken te doorzoeken. Vanwege de variërende bezoekkosten in een discreet dynamisch en een dynamisch netwerk is de Markov onafhankelijkheid van de optimale oplossing niet langer van kracht. Dit betekent dat een optimale oplossing van het totale probleem niet geconstrueerd kan worden uit een samenstelling van de optimale oplossingen van de deelproblemen. "Verdeel en heers" gaat niet op. Met andere woorden: in een treindienstregeling hoeft de optimale route van A naar C via B geen samenstelling te zijn van de optimale route van A naar B en de optimale route van B naar C, omdat er mogelijk geen aansluiting bestaat. Hierdoor kan geen van de traditionele kortste-pad algorithmen gebruikt worden. We hebben het algorithme voor het doorzoeken van discrete netwerken aangepast voor discrete dynamische en dynamische netwerken: we definiëren welke oplossingen van de deelproblemen vereist zijn om de optimale oplossing van het totale probleem te kunnen samenstellen.

Om de zoekefficiëntie te verhogen hebben we de Space Reduction Method (Ruimte Reductie Methode) ontwikkeld. In SRM worden eerst oplossingen beschouwd in een eenvoudigere zoekruimte, de abstractieruimte geheten, om delen uit de volledige zoekruimte te kunnen snijden. In een treindienstregeling bestaat de abstractieruimte uit een netwerk waarin alle treinen tussen twee stations zijn vervangen door één verbinding met een geschatte reistijd. Door dit abstracte netwerk te doorzoeken wordt bepaald welk deel van het dienstregelingsnetwerk waarschijnlijk de optimale oplossing bevat. SRM reduceert de zoekruimte zonder de optimale oplossing te verliezen. Zodra een oplossing is gevonden, controleert SRM namelijk of er misschien een betere oplossing buiten de zoekruimte zou kunnen bestaan. We laten zien hoe SRM toegepast kan worden op het doorzoeken van een discreet dynamisch netwerk.

Heuristieken kunnen worden gebruikt om de efficiëntie van zoekalgorithmen verder te verbeteren. We beschrijven hoe de resultaten van SRM gebruikt kunnen worden in een A* achtige uitbreiding van het algorithme om discrete dynamische netwerken te doorzoeken, door tijdens het zoeken de voorkeur te geven aan die punten die vermoedelijk dichter bij het doel liggen. SRM geeft voor ieder punt (station) in de zoekruimte een consistente onderschatting van de afstand (reistijd) die nog resteert naar het doel (aankomststation). Tijdens het zoekproces wordt deze schatting gebruikt om die punten te selecteren die vermoedelijk het dichtst bij het doel liggen.

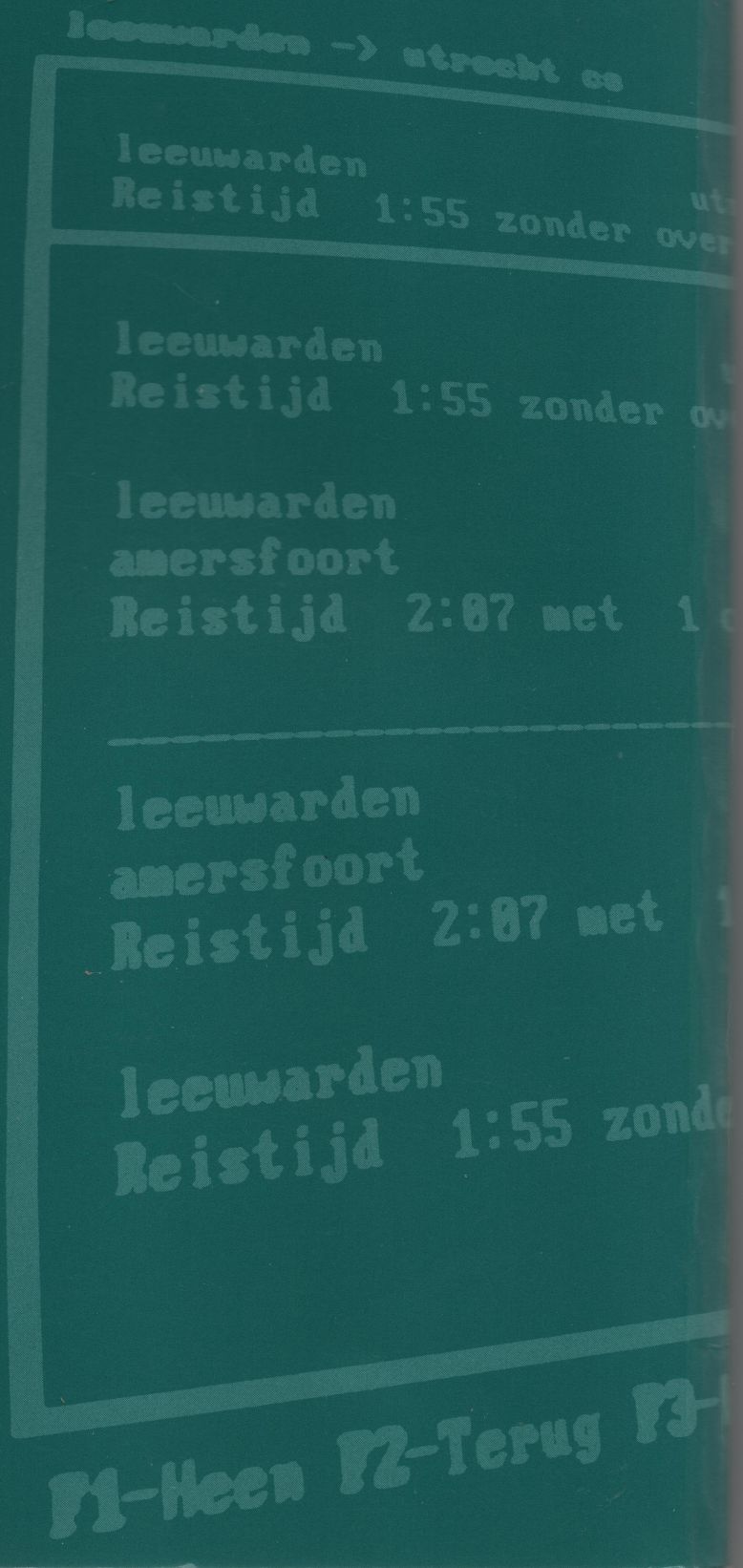
Een uitstekende manier om het zoeken naar een optimale oplossing te beperken is er voor te zorgen dat het te doorzoeken netwerk zo klein mogelijk is. Sommige punten kunnen uit het netwerk verwijderd worden als ze noch vertrekpunt

noch doel zijn. Bij een treindienstregeling geldt: als het geen zin heeft om over te stappen op een station, en als het bovendien niet om een vertrek- of aankomststation gaat, dan kunnen we dit station bij het zoeken buiten beschouwing laten. We laten zien hoe het algorithme voor het doorzoeken van discrete dynamische netwerken kan worden aangepast om met deze 'verborgen' punten om te gaan.

Indien een systeem wordt gebruikt om reizigers te adviseren over hun treinreis, dan is het niet voldoende alleen de snelste routes te geven. In de praktijk willen reizigers ook informatie over routes met zo min mogelijk overstappen. We beschrijven hoe de snelste route geoptimaliseerd kan worden naar het aantal overstappen, en hoe sommige (suboptimale) routes met minder overstappen kunnen worden gevonden door een tijdsboete voor overstappen te gebruiken. Door routes met overstappen tijdens het zoeken te beboeten, kunnen routes zonder of met minder overstappen worden gevonden.

Ofschoon de optimale of de snelste oplossing goed is gedefinieerd, is het veel minder duidelijk wat het *beste antwoord* op een vraag van een bepaalde gebruiker is. In de praktijk blijkt dat veel gebruikers hun vraag normaliter overspecificeren en dat deze vraag zelden volkomen vast ligt. Er zijn veel factoren die het 'beste' antwoord bepalen, en de meeste gebruikers kunnen deze factoren niet eens alle expliciet maken. Van treinreizen is het bekend dat het aantal keren overstappen belangrijk is, maar er kunnen meer factoren bijdragen tot het beste antwoord. Verder kunnen deze factoren van geval tot geval verschillen. Daarom is het niet mogelijk om het beste antwoord te definiëren in termen van doelen en eisen. Voor ons onbruikbaar zijn daardoor technieken die met meerdere doelen naar het kortste pad zoeken of die voor elk doel een suboptimaal pad vinden. In plaats daarvan zoeken we naar een aantal optimale oplossingen en suboptimale oplossingen met minder overstappen, en gebruiken een algemeen "gezond verstand" gebruikersmodel om al die oplossingen te selecteren die relevant zijn voor een gebruiker. De gebruiker zelf beslist welke oplossing het beste is.

Het algorithme voor het doorzoeken van discrete dynamische netwerken en de zojuist beschreven technieken zijn geïmplementeerd in een werkend systeem (TRAINS), dat de volledige Nederlandse treindienstregeling doorzoekt. We beschrijven hoe TRAINS werd geïntroduceerd als een hulpmiddel op telefonische informatiecentra van de Nederlandse Spoorwegen (NS). Daarna werd TRAINS aangepast voor algemeen gebruik en uitgebracht, eerst als relatiegeschenk, daarna als een officieel NS produkt (NS Reisplanner).


leeuwarden -> utrecht cs
leeuwarden
Reistijd 1:55 zonder over
leeuwarden
Reistijd 1:55 zonder ov
leeuwarden
amersfoort
Reistijd 2:07 met 1
leeuwarden
amersfoort
Reistijd 2:07 met
leeuwarden
Reistijd 1:55 zond
F1-Heen F2-Terug F3-